# NAVAL POSTGRADUATE SCHOOL

## Monterey, California


# THESIS

IMPLEMENTATION OF A SERIAL DELAY INSERTION TYPE LOOP COMMUNICATION FOR A REAL TIME MULTITRANSPUTER SYSTEM

by

Bekir Evin

June 1985

Thesis Advisor: Uno R. Kodres

Approved for public release; distribution is unlimited

# REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Implementation of a Serial Delay Insertion Type Loop Communication for a Real Time Multitransputer System

5. TYPE OF REPORT & PERIOD COVERED
Master's Thesis
June 1985

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Bekir Evin

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Naval Postgraduate School
Monterey, California 93943-5100

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Naval Postgraduate School
Monterey, California 93943-5100

12. REPORT DATE
June 1985

13. NUMBER OF PAGES
90

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

15. SECURITY CLASS. (of this report)

15a. DECLASSIFICATION/ DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Transputer, Occam, Concurrent Processing, Multiprocessing, Multiprocessor, Multitransputer, Multicluster, Real Time System, Loop, Delay Insertion Loop Interface

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
This thesis presents a design and implementation of a model of a multicluster loop interface. This Delay Insertion Loop type of interface is based on the IMS T424 Transputer and the Concurrent Sequential Processes type of probramming language OCCAM. The Loop-type of communications systems are described and the Delay Insertion type of interface has been selected as the most appropriate one for high performance real-time applications. The OCCAM programming language, hosted on the VAX 11/780 VMS system (continued)

(VAX-Virtual address eXtension, VMS-Virtual Memory System), was used to program the simulated version of the multicluster loop interface.

Approved for public release; distribution is unlimited.

# Implementation of a Serial Delay Insertion Type Loop Communication for a Real Time Multitransputer System

by

Bekir Evin
Lieutenant J.G., Turkish Navy
B.S., Turkish Naval Academy, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ENGINEERING SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1985

## ABSTRACT

This thesis presents a design and implementation of a model of a multicluster loop interface. This Delay Insertion Loop type of interface is based on the IMS T424 Transputer and the Concurrent Sequential Processes type of programming language OCCAM. The Loop-type of communications systems are described and the Delay Insertion type of interface has been selected as the most appropriate one for high performance real-time applications. The OCCAM programming language, hosted on the VAX 11/780 VMS system (VAX-Virtual Address eXtension, VMS-Virtual Memory System), was used to program the simulated version of the multicluster loop interface.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENT

I would like to acknowledge and thank my thesis advisor Prof. Uno R. Kodres for his encouragement and guidance in this thesis. His advice and suggestions often provided the needed incentive required to overcome difficult obstacles.

I would like to thank my friend Lt.J.G. Zafer Selcuk and the staff of the Computer Science Department for their help and understanding.

A special note of appreciation goes to my wife, Mine, because of her invaluable patience, encouragement and understanding during this intensive educational period at the Naval Postgraduate School, and whose assistance was a really significant contribution to this thesis. Also special appreciations go to my parents and my family. I would like to devote this thesis to all of them.

## DISCLAIMER

Many terms used in this thesis are registered trademarks of commercial products. Rather than attempt to cite each individual occurrence of a trademark, all registered trademarks appearing in this thesis will be listed below following the firm holding the trademark.

INMOS Limited Corporation, Colorado Springs, Colorado

INMOS
OCCAM
Transputer
IMS T424
Inmos links
IMS 1400 (RAM)
IMS 3630 (RAM)
IMS 2620 (EPROM)

Just for conceptual explanation purposes, some parts of the Chapter IV and V are reproduced by the following permission.

Cay Weitzman, DISTRIBUTED MICRO/MINICOMPUTER SYSTEMS: Structure, Implementation and Application, Copyright 1980, pp. 62-71, 253-255. Reproduced by permission of Prentice-Hall, Inc., Englewood Cliffs, N.J.

# I. INTRODUCTION

## A. BACKGROUND

As the computer technology grows rapidly, the complexity of computer systems together with hardware and software has been increasing. The lowest possible cost, smallest incremental expansion capability and the demand for enhanced user convenience have influenced the trend toward multiprocessor systems.

To enhance throughput, reliability, computing power, parallelism, and economies of scale, additional processors can be added to some systems. In early multiprocessor systems the additional processors had specialized functions, e.g., I/O peripherals. Later multiprocessing systems evolved to include the concept of one large CPU and several peripheral processors. These processors may perform quite sophisticated tasks, such as running a display. A more common type of multiprocessing is a system having two or more processors, each of equal power. [Ref. 1]

There is also the computer network, in which many different computers are connected to perform repetitive functions, often at great distances from one another. They typically perform functions by spreading the pieces of the function around the total system.

There are various ways to connect and operate a multiprocessor system, loop architecture being one of them. The primary advantage of loop systems are their relatively low cost and high modularity. Different loop types will be introduced and especially the Delay Insertion Loop will be emphasized.

The bit-serial interconnect scheme is attractive since it allows the user to interconnect processors made by the different manufacturers and often with different internal architectures without too much concern for the communications impact on the existing system software. [Ref. 2]

The Transputer (transistor computer) is a new single chip computer which is presented and used as a powerful uniprocessor in concurrent multiprocessor systems with a new programming language OCCAM.

## B. MOTIVATION OF THIS THESIS

The importance of communication in a multiprocessor system increases as the number of the processors in the system increases. The need for an effective communication for building a cluster of the processors lead us to choose loop type and especially the Delay Insertion Loop type of serial communication interface, since it provides efficient use of transmission facilities as well as signal transparency, expandability (growth flexibility), relatively low cost and high modularity.

The Transputer is chosen as a powerful component processor for building multiprocessor system, so the transputer is designed to implement a particular programming language, OCCAM, efficiently. OCCAM enables the behaviour of concurrent systems to be explicitly programmed and controlled. The OCCAM language retains the efficiency, in terms of program density and performance, of an assembler, while offering the productivity and reliability advantages of programming in a high level language. [Ref. 3]

If we look at some quantitative information about the performance and capacity of the transputer, IMS T424 [Ref. 4], it can be understood why we are motivated for this work.

TABLE 1

CAPABILITIES OF THE IMS T424

| | |
|---|---|
| processor . . . . . | 32 bits |
| processing speed . . . | 10 MIPS (950 nanosecond mult.) |
| memory capacity . . . . | 32 bit address bus |
| built in memory . . . . | 4 KBytes RAM |
| serial bus . . . . . | 4 INMOS links (1.5 Mbytes/sec) |
| parallel bus . . . . . | 25 Mbytes/sec (max. transfer) |
| peripheral interface . | 8 bits bidir. (4 Mbytes/sec) |
| power dissipation . . . | 0.9 Watts |
| physical . . . . . | 45 $mm^2$ chip mounted in an 84 contact leadless chip carrier. |

Table 1 shows attractive parameter values of the transputer as a uniprocessor component to use in a multiprocessing environment. The transputer provides excellent hardware for implementing concurrent processing. It is designed using a reduced instruction set architecture which implements the OCCAM concurrent programming language efficiently. [Ref. 5]

Another important feature of the transputer is the four bidirectional serial communication channels. It is possible to obtain multiple communication paths between two elements of the multitransputer system by appropriately connecting the serial links. These multiple paths provide for the graceful degradation for redundant multitransputer systems. The failed element can be simply by-passed in the multisystem and processing continues with other elements of the system.

## C. OBJECTIVES

This thesis implements a model of a Delay Insertion Loop type of serial communication interface for a real-time multitransputer system by using the programming language OCCAM. A four transputer model will be used to illustrate the interface programming. Although the model uses only four transputers, any number of transputers may be connected together using this type of loop concept. Fault tolerance issues and how these issues can be resolved are discussed separately.

## D. THESIS ORGANIZATION

The introduction just presented is designed to provide the reader with a brief look at a multitransputer architectural concept, the transputer and OCCAM language.

Chapter II will present the hardware architecture and the capabilities of the transputer and multitransputer systems. Chapter III will present the OCCAM as a new concurrent processing language and its special features. Chapter IV will describe multitransputer systems, multiprocessing, interconnect structures and loop type communication systems. Chapter V will implement a four transputer model of a Delay Insertion Loop Type of serial communication interface.

And the final chapter will present conclusions, observations that resulted from this thesis effort and suggestions for further research. The software program of the system is provided as an appendix.

## II. HARDWARE

### A. TRANSPUTER

The transputer is a programmable component. The term "Transputer" is derived from 'transistor' and 'computer', since the transputer is both a computer on a chip and a silicon component like a transistor. As a transistor computer, it is a single chip computer which provides a direct implementation for the process model of computing, in which each process is an independent computation with its own data and program. The processes are executed in a time shared mode on the transputer and special instructions are provided to support the process model of communication. A transputer is a microcomputer with its own local memory and with link interfaces for connecting one transputer to another transputer [Ref. 3].

### B. WHY TRANSPUTER ?

There are some problems in the design of concurrent systems. Three apparent ones are :

1. Hardware problems (How to connect the computers).
2. Programming problems (How to program tens or hundreds of connected machines).
3. Design problems (How to design the system as a whole). [Ref. 6]

IMS T424 32 bit transputer provides an effective solution to these problems. The programming problem may be solved by the use of an appropriate concurrent programming language; OCCAM is recommended.

The transputer can be used as a single chip stand-alone component or in networks to build high performance concurrent systems. As a stand-alone component, the transputer can be programmed in conventional high level languages. It is designed to implement a particular concurrent programming language, OCCAM, efficiently.

The transputer allows arbitrarily large systems to be constructed using localized processing and communication. Its locality is exploited by OCCAM and multitransputer systems can be used effectively.

The transputer uses point-to-point serial communication links, therefore it provides maximum communication speed with minimal wiring. Correspondingly, OCCAM uses point-to-point channels.

## C. GENERAL FEATURES OF TRANSPUTER

The main components of the transputer : memory, processor, links and peripheral interface will be described in the following subsections. [Ref. 3]

### 1. Memory

T424 contains 4 Kbytes of static RAM which cycles synchronously with the processor and provides maximum data transfer rate of 80 Mbytes/sec, Figure 2.1 . The memory interface uses a 32 bit multiplexed data and address bus to give high performance access to external memory. It can extend internal address capability to a total of 4 Gbytes in a single linear address space.

A number of preset timing configurations is provided to suit a wide variety of memories. All the timing strobes are generated for dynamic RAM's as well as the necessary refresh cycles. A memory cycle consists of six phases. Each refresh cycle outputs a nine-bit refresh address and the

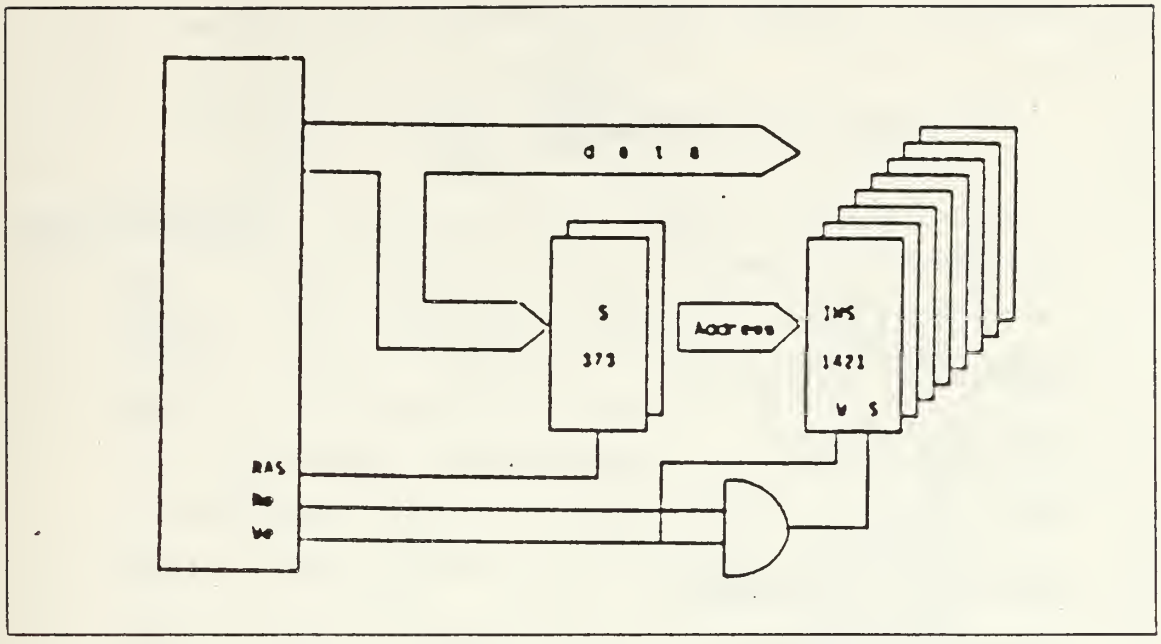


Figure 2.1 Memory Interface Driving Static RAM's

user can choose the interval between refresh cycles. An asynchronous wait input is provided so that the memory timing can also be determined externally if required. Wait states generated by the configurable strobes can extend the interface cycle for slow external devices. Wait states generated by external logic can extend the memory cycle indefinitely. The cycle to access memory is completed in 150 nano seconds providing data rate of 25 Mbytes/second maximum, without requiring the phases for address multiplexing.

2. Processor

The T424 32 bit processor is designed to implement high-level languages (e.g. OCCAM, C and Pascal) efficiently and to provide high performance communication between concurrent processes. Its instruction execution rate is 10 MIPS. Typical instructions execution times are shown in Table 2 .

TABLE 2

INSTRUCTION EXECUTION TIMES OF IMS T424

| INSTRUCTION | EXECUTION TIME (nano second) |
|---|---|
| arithmetic operands | |
| + , - | 50 |
| multiplication | 950 |
| division | 1950 |
| remainder | 1950 |
| comparison operators | |
| =, ≠, >, ≥, <, ≤ | 100 |
| logical operators | |
| AND , OR | 50 |
| shifting | |
| <<[n], >>[n] | 50n+50 |
| identifiers | |
| variable | 120 |
| vector variable | 160 |
| expression evaluation | |
| constant | 70 |
| parenthesis | 50 |
| constructor | |
| sequential | 0 |
| parallel | 450n-200 |
| alternative | 600n+600 |
| branch (IF) | 150n |
| repetitive (WHILE) | 200 |
| primitives | |
| ! (output) ? (input) | 625 |
| assignment | 0 |

High-level language expressions are evaluated on an evaluation stack of 32-bit registers. The instructions specify the registers of the evaluation stack implicitly, allowing compact coding of instructions. The correct and optimal sequence of these instructions is easy for a compiler to generate. Each instruction is one byte long and divided into two four bit fields: function and operand. It is also simple to decode, which contributes to the high performance of the processor. High-level language support is enhanced with instructions for array bounds checking, arithmetic overflow detection and support for multi-word-length arithmetic.

The processor provides direct support for the OCCAM model of concurrency and communication. It has a scheduler which enables any number of concurrent processes to be executed together, sharing the processor time. Process communication is implemented by memory-to-memory block move operations. These fully utilize the bandwidth available from the on-chip RAM. The small number of registers which form the process context and the use of on-chip RAM combine to provide a fast process switch time.

In concurrent processing, the uniprocessor executes programs sequentially. It implements parallel processes by sharing its time between the set of processes which are active at any instant. A process is active when it is not waiting for input or output. When communication happens, the currently executing process is set inactive to wait for communication and the next process on the active queue starts to execute. When a communication channel becomes ready, the message is passed and the waiting process is linked to the end of the active process queue. The process continues to execute, whenever its turn in the queue comes up.

The T424 processor supports two levels of priority. High priority processes can be used for message through-routing or for fast response to external events. PRIPAR (priority parallel) process may have two components. A queue of active processes is maintained for each priority level. A priority 1 (low priority) process is executed whenever there are no active priority 0 (high priority) processes. If there are no active priority 0 processes, the latency (time from an external channel becoming ready to the start of its first instruction of the relevant waiting priority 0 process) is typically 600 ns (maximum 2600 ns). Otherwise, if a priority 0 process is already executing, the relevant waiting process is linked to the end of the priority 0 queue.

The processor also supports the OCCAM concept of alternative inputs, which allows hardware and software interrupts to be programmed in a high-level language.

The processor includes a timer: a process can read the time or can wait until the time reaches a value.

The processor sees memory as a linear signed address space of 4 Gbytes ($2^{32}$ bytes), with no difference between on-chip and off-chip memory except for performance. The signed address space allows address calculations to be handled in the same way as arithmetic calculations. This not only simplifies the processor (and compiler) design but also means that arithmetic overflow can be treated uniformly.

The processor bootstraps itself either from program in external ROM or from any of the INMOS serial links. Any error detected by the processor can drive an error signal which can be used to stop the processor so that the error is contained and the cause of the error can be analysed.

As an external memory cycle can be used either to access one data word or to fetch four instructions, most programs require more memory accesses for data than for the program. In general, therefore, better results are obtained by placing program off chip rather than placing data off chip. If both the program and local variables can be held on chip, so that most memory accesses are to on-chip RAM, the performance will be close to the performance of all program and data on chip.

A high priority process running in the transputer takes priority over all low priority activity, including communication. Communication to a high priority process occurs concurrently with another high priority process running in the transputer. For a 50 ns cycle time T424 processor (Table 3), if all processes are running at the same priority, and all the links are transferring at 10

Mbits/s in both directions at the same time, using internal memory, the maximum interference on processor performance is about 8 %. The average interference is negligible.

The overall size of a program is given by the sum of the sizes of its program elements. All timing averages and the maximum program execution time are given by the sum of the execution times of the individual program elements. If the program is held in the external memory, the external program fetch time must be added to obtain the program execution time. If data is held in external memory, the external data access time must also be added to obtain the program execution time. The processor shares memory cycles with its input/output interfaces. Each concurrent access by an interface channel delays the processor by an average of 30 ns. The maximum reduction in performance is 10 %; under typical conditions the reduction is negligible.

For integer computations, the IMS T424 transputer is nearer a dedicated digital signal processing device than a mainstream 32 bit microprocessor. However, in common with other designs, it does not provide built-in floating point operations. The design of its instruction set, including appropriate shifting operations, enables an efficient software implementation of IEEE floating point specification, comparable in speed with an established floating point coprocessor. A software library implementing both 32 and 64 bit floating point will be available from INMOS during 1985, and is likely to satisfy many applications needs. Using appropriate sequences of instructions, the T424 can perform arbitrary length integer arithmetic, real arithmetic, fractional arithmetic and fixed point arithmetic. An obvious possibility for a new transputer product is one with embedded floating point capability. [Ref. 7]

3. Links

The IMS T424 transputer has four standard INMOS serial links which provide high speed intercommunication between transputer products and enable a rich variety of networks to be constructed. The link interfaces and the processor all operate concurrently and each link interface operating independently provides block message transfers to and from the memory of the transputer.

Each autonomous link interface has an output and an input signal, both of which are used to carry data and protocol bits. A message is transmitted as a sequence of bytes. After transmitting a data byte, the sending transputer waits until an acknowledge has been received, signifying that the receiving transputer is ready to receive another byte, before transmitting the next byte. Each link implements two OCCAM channels. The protocol allows the receiving transputer to transmit an acknowledge as soon as it starts to receive a data byte and provides end-to-end channel synchronization. This asynchronous protocol guarantees reliable transmission in spite of possible delays in either the sending or receiving transputer. A message transmission via a link to or from a process executing on the T424 is performed by an autonomous block transfer engine. The process itself is descheduled during the transfer, allowing the transputer processor to execute other processes. During transmission of a message, both sending and receiving processes will be set inactive, and they will only be linked to the end of their respective active queues after the final byte has been acknowledged.

Table 3 shows different speeds of the IMS T424 transputer.

The links support a universal standard bit rate of twice the input clock frequency (10 Mbit/s with a 5 MHz

TABLE 3

SPEED OF IMS T424

| TYPE | INSTRUCTION THROUGHPUT | PROCESSOR CLOCK SPEED | PROCESSOR CYCLE TIME | INPUT CLOCK |
|---|---|---|---|---|
| IMS T424-10 | 5 MIPS | 10 MHz | 100 ns | 5 MHz |
| IMS T424-12 | 6 MIPS | 12.5 MHz | 80 ns | 5 MHz |
| IMS T424-15 | 7.5 MIPS | 15 MHz | 67 ns | 5 MHz |
| IMS T424-17 | 9 MIPS | 17.5 MHz | 57 ns | 5 MHz |
| IMS T424-20 | 10 MIPS | 20 MHz | 50 ns | 5 MHz |

input clock). All transputers, of whatever word length and speed selection, support the universal communications frequency as a product range standard. An internal link clock is derived from the input clock and data bits are transmitted synchronously with this clock. Data reception is asynchronous.

As shown on Table 3, the maximum speed of the IMS T424 transputer is 20 MHz providing 10 MIPS of throughput. The data rate on each link can be programmed, using link set configuration channel [Ref. 4]. 20 Mbits/s gives a maximum data rate of 1.8 MBytes/s on a channel.

4. Peripheral Interface

The peripheral interface is an 8 bit bidirectional bus which may be used to input and output sequences of bytes. It provides access to industry standard devices such as eight bit parallel controllers for auxiliary memory. A block message transfer capability between memory and the peripheral interface is provided by the interface controller. There are two control lines which may be used to address external devices, and an "Event" input to provide an interrupt capability.

The "Event" input may be used to communicate with waiting processes and hence cause it to be scheduled. This provides an input functionally similar to an interrrupt, in a manner consistent with the process model of the transputer. The typical latency for this interrrupt is 600 ns. The "Event" input can also be used to enable the peripheral interface to respond to being accessed from a standard microprocessor bus.

The interface is accessed via four standard input and output channels. All eight channels use the same 8 bit path and transfer handshake, with the processor initiating the transfer. The transfers are synchronized to a separate external clock, which need not have any fixed relationship with the transputer input clock. Asynchronous operation is also permitted, but at a lower speed than for synchronous operation.

Externally addressable devices may be connected via the peripheral interface. For instance, by using one output channel as the address channel, another as the write data channel, and one input channel as the read data channel. Both addresses and data may be arbitrarily long sequences of bytes. The 4 Mbytes/s data rate provided by the interface allows the connection of high performance peripheral chips, without the need for FIFO's or DMA controllers.

# III. SOFTWARE

## A. OCCAM

As a new programming language, OCCAM, is designed in conjunction with the INMOS transputer [Ref. 8]. It supports concurrent applications in which many parts of a system operate separately and interact. OCCAM can capture the hierarchical structure of a system by allowing an interconnected set of processes to be regarded from the outside as a single process. At any level of detail, the programmer is only concerned with a small and manageable set of processes.

## B. WHY OCCAM ?

The novelty of OCCAM is in its treatment of concurrency. OCCAM enables the behaviour of concurrent systems to be explicitly programmed and controlled. It also gives the efficiency, in terms of program density and performance, of an assembler, while offering the productivity and reliability advantages of programming in a high level language.

OCCAM enables the programmer to express a program in terms of concurrent processes which communicate by sending messages through communication channels. This has two important consequences. First, it gives the program a clear and a simple structure as the individual processes operate largely independently. Second, it allows the program to exploit the performance of many computing components, as each concurrent process may be executed by an individual processor.

OCCAM provides a methodology for designing present and future concurrent systems using transputers in just the same way that Boolean Algebra provides a methodology for designing today's electronic systems from logic gates.

The task of the system designer is eased because of the architectural relationship between OCCAM and the transputer. A program running in a transputer is formally equivalent to an OCCAM process, so that a network of transputers can be described directly as an OCCAM program. [Ref. 3]

## C. GENERAL FEATURES OF OCCAM

### 1. Processes

A process starts, performs a sequence of actions, and then terminates. Each action may be an assignment, an input, an output or SKIP (Table 4). An assignment changes the value of a variable, an input receives a value from a channel and an output sends a value to a channel. The process SKIP has no effect. The process STOP starts but never proceeds, its main use is to prevent an erronous process from proceeding. At any time between start and termination a process may be ready to communicate on one or more of its channels. Each channel provides a one way connection between two concurrent processes; one of the processes may only output (write) to the channel, and the other may only input (read) from it.

A process may be ready and waiting to input from any one of a number of channels. In this case, the input is read from the first channel which is used for output by another process. Communication is synchronous. The value to be transmitted is copied from the output process to the input process when both an input process and an output process are ready to communicate on the same channel.

OCCAM may be used to program an individual transputer. The transputer shares its time between the concurrent processes, and the channels are implemented by values transmitted in the main memory.

An OCCAM program may be executed by a network of transputers. Nevertheless, the same program may be executed unchanged by a smaller network or even by a single transputer. Each transputer with local storage executes a process with local variables, and each connection between two transputers implements a channel between two processes.

Three primitive processes, as mentioned above, are input, output and assignment. OCCAM programs are built from these three primitives given in Table 4. They can be combined sequentially or concurrently to create more complex processes, and so they form the building blocks for a program.

TABLE 4
PRIMITIVE PROCESSES OF OCCAM

| PRIMITIVES | SYNTAX |
|---|---|
| ASSIGNMENT | variable := expression |
| INPUT | channel ? variable |
| OUTPUT | channel ! variable |

a. Assignment

An assignment is indicated by the symbol ':='. It transfers the value of its expression to the named variable. The expression is evaluated and the variable is set to the resulting value, then the assignment process terminates. The variable may be a simple variable or an element of a vector of variables selected using either byte or word subscripts.

An assignment 'y := e' sets the value of the variable y to the value of the expression e and then terminates. For example, 'y := 0' sets y to zero, and 'y := y + 1' increases the value of y by 1.

b. Input

An input process reads (receives) a value from the channel into a variable. The '?' symbol denotes the input process. This primitive reads a value from the specified channel. It provides synchronization with a concurrent process, which places a synchronizing signal on the same channel. An input primitive sets the value of a variable to a value read from a channel. The input primitive waits until an output primitive using the same channel is executed in parallel with the input.

An input 'c ? v' reads a value from the channel c, and assigns it to the variable v and then terminates. An input 'c ? ANY' reads a value from the channel c, and discards the value.

A multiple input is equivalent to a sequence of separate input processes for each variable in turn, in left to right order. Each input is separately synchronized with an output process being executed in parallel. Each variable may be a simple variable, or a word or byte subscripted element of a vector of variables.

c. Output

An output process writes (sends) the value of the expression to the channel. An output is indicated by the symbol '!'. An output waits until an input using the same channel is executed. It then outputs the value of the expression to the channel and terminates. A multiple output is equivalent to a sequence of outputs, which writes the value of each expression in turn, in left to right order.

Each output is separately synchronized with an input process executed in parallel.

An output 'c ! e' writes the value of the expression e to the channel c. An output 'c ! ANY' writes an arbitrary value to the channel c.

## 2. Constructs

A number of processes may be combined to form a sequential, parallel, conditional, alternative, repetitive, replicative construct. A construct is itself a process, and may be used as a component of another construct. Each component process of a construct is written two spaces further from the left hand margin, to indicate that it is part of the construct.

### a. Sequential

It is necessary to do a number of steps one after another in many applications. Figure 3.1 shows the flow diagram of this sequential construct. The component processes are executed one after another in this structure.

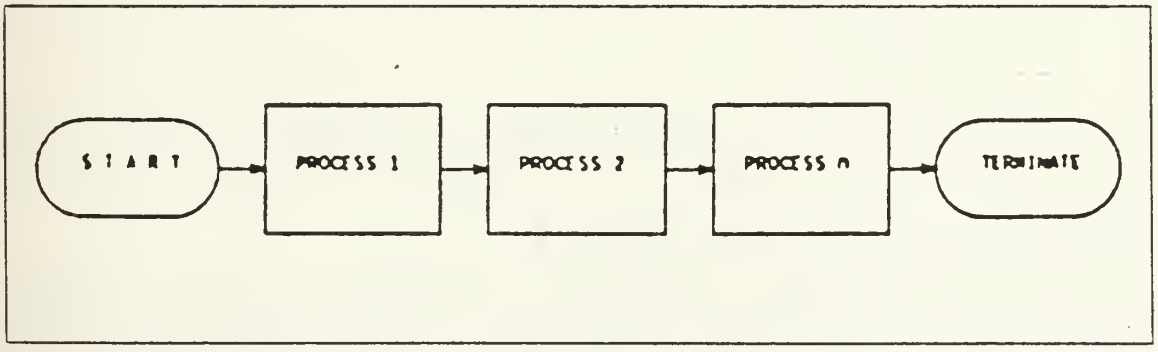


Figure 3.1 Flow Diagram of the Sequential Construct

A sequential process takes the form of the keyword SEQ followed by the component processes, each on a

new line, all at an extra level of indentation as shown in Table 5.

TABLE 5

SEQUENTIAL CONSTRUCT

```
SEQ                          SEQ
  process 1                    c1 ? x
  process 2                    x := x + 1
    .                          c2 ! x
    .
  process n
```

The component processes, process 1, process 2, ... process n are executed one after another. Each component process starts after the previous one terminates and the construct terminates after the last component process terminates. For example, a sample SEQ construct given in Table 5, reads a value, adds one to it, and then writes the result.

SEQ and its component processes can be regarded as a single process.

b. Parallel

If it is required many processes to be running as a concurrent system, a parallel process can be constructed as shown in Figure 3.2.

As seen in Table 6, the keyword PAR is followed by a number of component processes, each starting on a new line and indented. Then the effect is to execute all of the component processes together, which is achieved by sharing the processor time between the set of active processes.

The parallel construct terminates after all the component processes are terminated. If there is no component process, the construct terminates immediately.

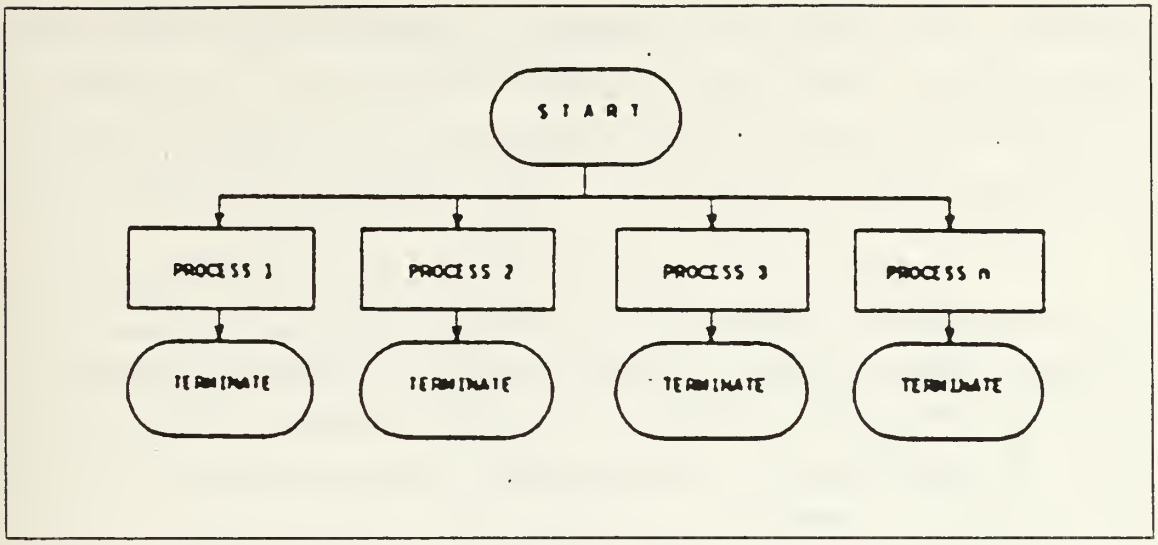


Figure 3.2 Flow Diagram of the Parallel Construct

TABLE 6

PARALLEL CONSTRUCT

```
PAR                          PAR
  process 1                    c1 ? x
  process 2                    c2 ! y
     .
     .
     .
  process n
```

For example, if we have a parallel construct as seen in Table 6, two component processes are executed together, and are called concurrent processes. This sample construct allows input to x and output from y to take place together.

Concurrent processes communicate using channels. When an input from a channel c, and an output to the same

channel c are executed together, communication takes place when both the input and the output are ready. The value is assigned from the writing process to the reading concurrent process, and execution of both concurrent processes then continues.

Variables are not used for communication between the component processes of a parallel construct. However, a variable may be used in two or more component processes, provided that no component process changes its value by input or assignment. Two component processes of a parallel construct may communicate by sending values using a channel. One contains outputs to the channel, and the other contains the inputs from the channel. The processes are said to be connected by the channel. No other component of the parallel construct may use the same channel.

c. Conditional

A conditional construct takes the form of a conditional expression followed by a process, and it is able to execute if the expression evaluates to TRUE. As shown in Table 7, a conditional construction takes the form of IF followed by component conditionals. The construct is able to execute if one of its component conditionals is able to execute.

Process 1 is executed if conditional expression 1 is TRUE, otherwise process 2 is executed if conditional expression 2 is TRUE, and so on. Only one of the processes is executed and the construct then terminates. If there is no component able to execute, the construct terminates without any effect. Figure 3.3 shows the flow diagram of the conditional construct. A sample of this construct in Table 7, increases n only if the value of e is 0.

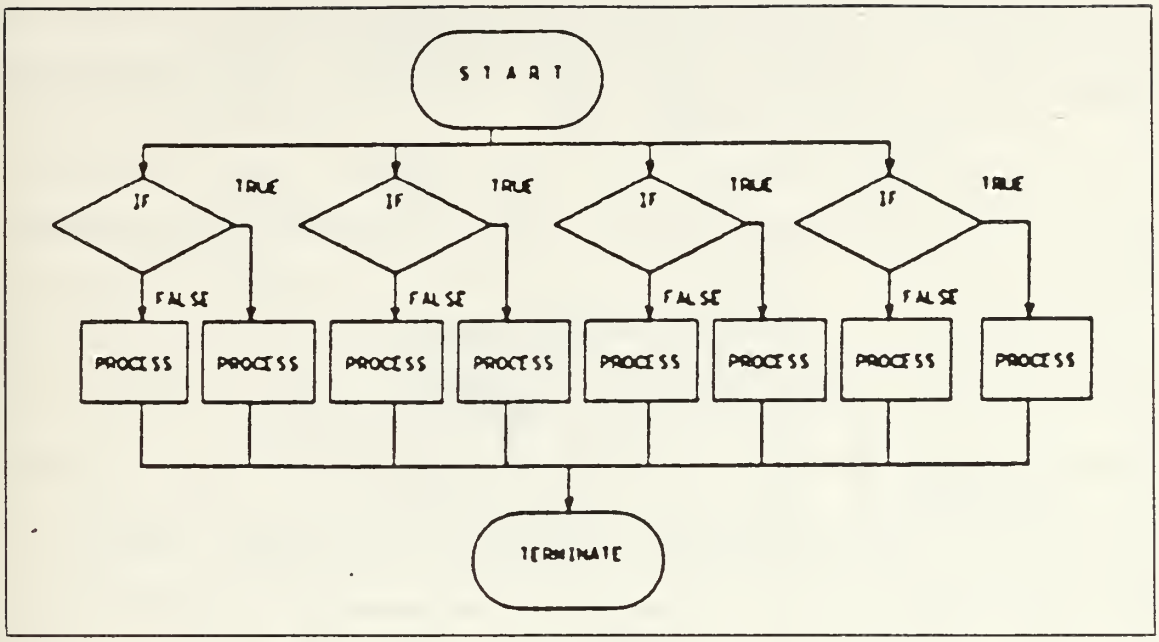


Figure 3.3 Flow Diagram of the Conditional Construct

TABLE 7

CONDITIONAL CONSTRUCT

```
IF                                          IF
  conditional expression 1                    e = 0
    process 1                                   n := n + 1
  conditional expression 2                    TRUE
    process 2                                   SKIP
      ...
```

d. Alternative

Sometimes a process has a number of channels associated with it and needs to perform one of a number of actions depending on which channel first sends it a message. This is achieved using the alternative construct, Figure

3.4, which chooses just one of its inputs for execution. The keyword ALT followed by a guarded process represents this construct as shown in Table 8.

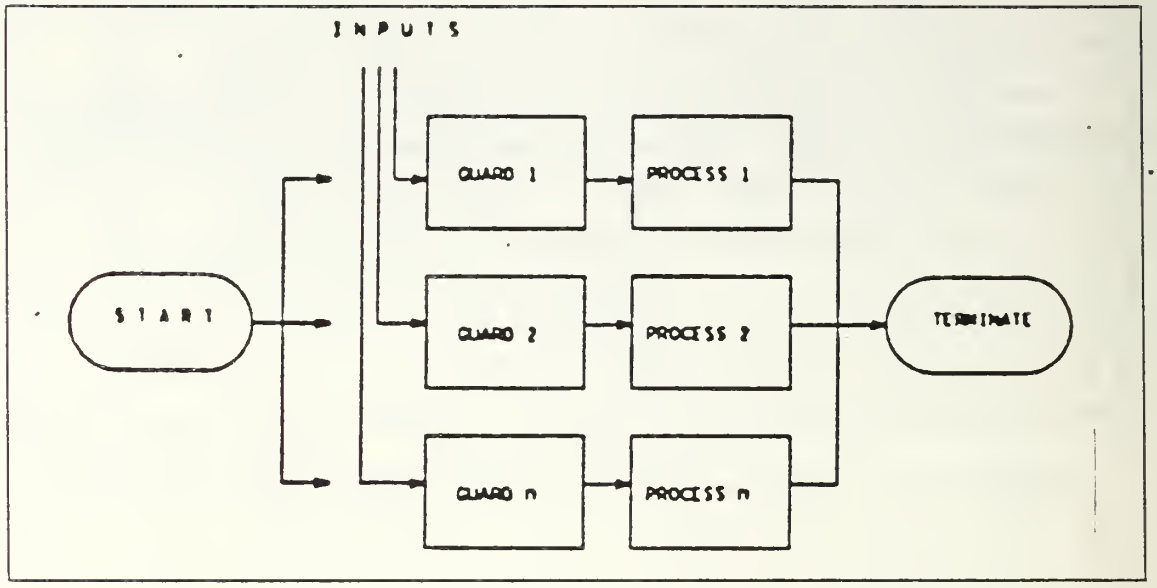


Figure 3.4 Flow Diagram of the Alternative Construct

TABLE 8

ALTERNATIVE CONSTRUCT

```
ALT
  guard-process 1
    process 1
  guard-process 2
    process 2
      .
      .
```

```
ALT
  index ? ANY
    number := number + 1
  sum ? ANY
    SEQ
      out ! number
      number := 0
```

An alternative process waits until one of guarded processes (inputs) is ready to execute. One of the ready guarded processes is then selected and executed. The construct then terminates. A guarded-process (input) starting with an input from a channel is ready if an output process is waiting to write to the channel. If the guarded process is selected, the component process is executed. If a guard contains an expression followed by an input or wait, the guarded process is ready only if both the value of the expression is TRUE and input or wait is ready. If a guarded-process is itself an alternative construct, then it is ready if one or more component guarded processes of the alternative construct is ready. If more than one guarded process becomes ready at the same time, an arbitrary one is selected. This may occur if the guarded processes contain inputs on the same channel.

For example, a sample construct in Table 8, either reads a signal from the channel index and increases the variable number by 1, or alternatively reads from the channel sum, and outputs the current value of the number from the channel out, and resets it to zero.

e. Repetitive

The repetitive construct takes the form of the keyword WHILE followed by a conditional expression, followed by a single component process indented on the next line. As shown in Figure 3.5, repetition construct repeatedly executes the process until the value of the condition is FALSE.

The component process in the repetition construct (Table 9) is executed as long as the expression is TRUE, and the construct terminates. If the conditional expression is initially FALSE, the process is not executed and the construct terminates right away. For example, a

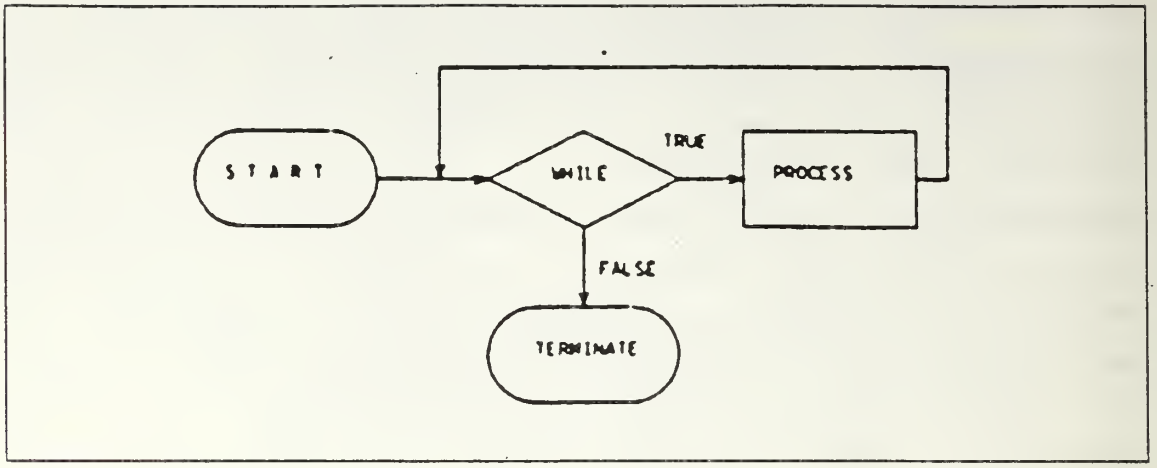


Figure 3.5 Flow Diagram of the Repetitive Construct

TABLE 9

REPETITIVE CONSTRUCT

| | |
|---|---|
| WHILE conditional expresssion<br>process 1 | WHILE a < 0<br>a := 0 - a |

repetition in Table 9, converts negative numbers to positive.

f. Replicator

A replicator is used with a constructor to replicate the component process a number of times (Table 10). Figure 3.6 shows the flow diagram of replication.

A replicator can be used with SEQ to provide a conventional loop. For example, 'SEQ i = [ 0 FOR n ]' causes the process to be executed n times.

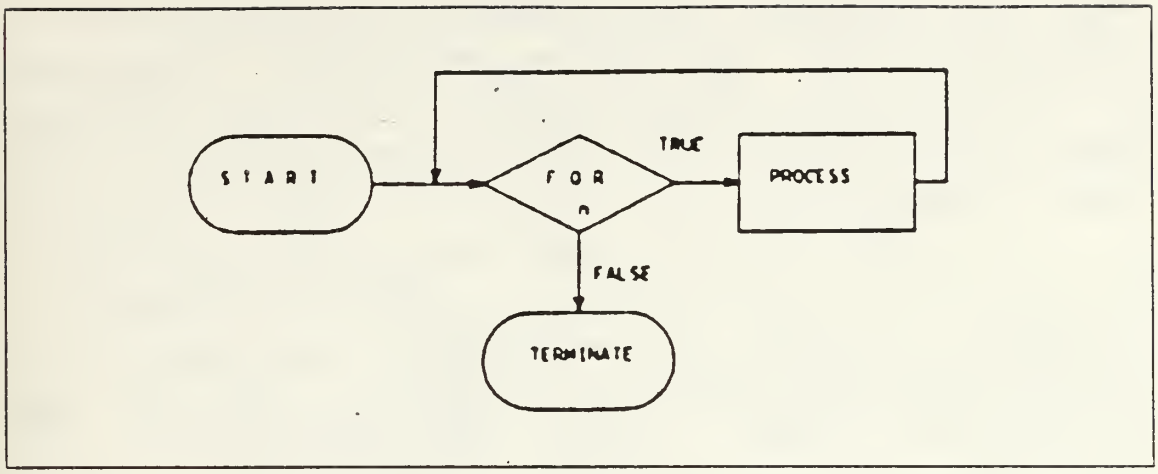


Figure 3.6 Flow Diagram of the Replicator Construct

TABLE 10

REPLICATOR CONSTRUCT

| SEQ i = [ base FOR count ]<br>process | PAR i = [ base FOR count ]<br>process i |
|---|---|

Replication can be used with PAR to construct an array of concurrent processes. For example, 'PAR i = [ 0 FOR n ]' constructs an array of n similar processes. The index i takes the values 0,1,...,n-1, in process 0, process 1 ... process n-1 respectively.

Replicator construct can also be used with ALT for reading from an array of channels.

The replicator declares an identifier to be the replicator index such as i, giving its base value and a count of the number of replications required. Its effect is to form a sequential, parallel, alternative or conditional

construct containing count components by replicating the component process, substituting successive integer values for the replicator index (starting at base). The substituted value for replicator index in the last component will be (base + count) - 1.

The replicator index can be used in expressions but not constant expressions, it may not be changed by assignment or input. An implementation may restrict the values of base and count to be constants, particularly when a replicator is used to form a parallel construct. If a count evaluates less than zero or equal to zero, then an empty construct is generated. This has the effect of termination for sequential, parallel and conditional constructs, and the effect of never being ready to execute for alternative processes.

### 3. Declaration Types

Every variable, expression and value has a type, which may be a primitive type or an array type. The type defines the length and interpretation of values of the type. Table 11 shows the primitive types which are available in all implementations.

TABLE 11

DECLARATION TYPES

| | |
|---|---|
| DEF | constant |
| VAR | variable |
| CHAN | channel |
| BOOL | boolean |
| BYTE | vector |
| INT | integer |

Array types are constructed from component types. For example, [n] T is an array type constructed from n components of type T. A channel type is either CHAN, or an array type in which every component type is a channel type. For example, 'CHAN x :' declares x as a new channel.

4. Named Processes

A process (procedure or subroutine) may be given a name. For example, Table 12 shows a sample process, it defines the named process square.

TABLE 12
A NAMED PROCESS

```
PROC square (INT n,sqr) =
  sqr := n * n  :
```

Process square is called with its name and actual parameters. For example, a call 'square (x,sqrx)' causes 'sqrx := x * x'.

5. Expressions

An expression is constructed from operators, variables, numbers, the truth values TRUE and FALSE and the brackets (and).

The boolean operators AND, OR and NOT operate on boolean values and yield boolean results.

The arithmetic operators +,-,*,/ and \ yield the arithmetic sum, difference, product, quotient and remainder respectively. Both operands must be of the same integer or real type.

The operators /\,\/ and >< operate on integers and yield the bitwise and, or, and exclusive or operations of the operands respectively.

The relational operators yield a result of type boolean, and both operands must be of the same type. The relational operators = and <> operate on any primitive type, and represent equals and not equals. The operators >,<,>= and <= operate on integers and reals, and represent greater than, less than, greater than or equals to, and less than or equals to.

Type conversion may be performed by using one of the type conversion operators $, $ROUND, and $TRUNC.

A string is represented as a sequence of ASCII characters enclosed in double quotation marks ". If the string has n characters, then it is an array of type [n] BYTE.

## 6. Configuration

Configuration is simply the allocation of processing resources to concurrent processes in a program. It is used to meet speed and response requirements by distributing programs over separate, interconnected computers, and by placing and prioritizing processes on single computers. Configuration does not affect the logical behaviour of a program. Simple implementations may omit or ignore some or all the configuration facilities. However, it does enable the program to be arranged to ensure that performance requirements are met.

Every computer has a local memory and a set of numbered ports. A physical connection between two computers connects a port on one computer to a port on the other computer. This implements up to two channels between the computers, one in each direction.

a. Prioritized Parallel

A parallel construct may be configured for a single transputer. The transputer shares its time between the component processes, and the channels are implemented by values in store. Therefore, OCCAM also contains the prioritized parallel construct declared as PRIPAR in addition to the regular parallel construct. This construct provides a different priority for each component process. Each component process of a PRIPAR construct is executed at a separate priority. The first process has the highest priority, the last the lowest. If P and Q are two concurrent processes with priorities p and q such that p<q, then Q is only allowed to proceed when P cannot proceed. An implementation may restrict the number of components which a prioritized parallel construct can have. And also an alternative construct can be used to provide the prioritized input primitives.

On any individual transputer, the outermost parallel construct may be configured to prioritize its components. A prioritized parallel (PRIPAR) construct ensures that a higher priority process always proceeds in preference to a lower priority one. The progress of a higher priority process is not affected by any lower priority one, except by communication on connecting channels. If several concurrent processes at the same priority are able to proceed, each one is given an opportunity to proceed in turn. The T424 transputer supports two levels of priority, priority 0 (high priority) and priority 1 (low priority).

b. Placed Parallel

A parallel construct may be configured for a network of transputers by using the PLACED PAR construct. Each component process (termed a placement) is executed by a

separate transputer. Port allocations are used to allocate channels to ports. The variables used in a placement must be declared within the placement. The values of the timer on different transputers are unrelated. A parallel construct configured for a network may be reconfigured for an individual computer.

## IV. MULTIPROCESSOR-MULTITRANSPUTER SYSTEM

A "pure" multiprocessor contains two or more processors of approximately comparable capabilities, which share access to all of memory and all input-output (I/O) channels, control units, and peripheral devices. The entire system is controlled by a single operating system. More frequently, multiprocessor systems share only portions of memory and input-output channels. The processors have dedicated memories for storing programs and local data and share data in segments of common memory.

Multiprocessor systems usually make it easier for the user to access the system, they generally provide increased performance through resource sharing, and they often increase the availability of a system. Multiprocessing systems can provide adaptability and rapid reconfiguration with the system functioning at different times as a very large and complex problem solver or as a network of smaller machines each dedicated to a unique task, or as something in between. A network of microprocessors can quite often duplicate the capability of one large expensive system at lower cost. They can also provide increased reliability since the total system can continue to operate despite individual processor failures, albeit with reduced capabilities, provided that some of the links between the processors remain intact. Also, since redundancy can be achieved at a lower cost using processors distributed over a large area, the survivability of the system, particularly in military applications can be increased. Furthermore, a distributed processing system can provide increased, distributed power and responsiveness because it can be closely tailored to the application. Additional multiprocessor systems can be

provided as needed, to ensure proper response time. Multiprocessor systems can also be designed to be cost effective when applied to a wide variety of applications, where the number of processors can be determined by the distributed processing requirements. A properly designed distributed processing system threatened by overload can be incrementally expanded by simply adding more processors.

Because of the above advantages, a large number of applications of multiprocessing systems can be seen such as control of electric power generation, distribution, and consumption, nuclear power processing facilities safeguard and control, health care delivery in hospitals and medical centers, climate control, security, waste disposal, many fire protection in large buildings, and in defense systems.

The disadvantages may or may not outweigh the advantages, depending on the system-unique requirements. On the minus side, the designer may be faced with increased software complexity. Application software may be more costly to develop for a distributed rather than a centralized system. In contrast to a single central processor based system with only one executive, a distributed system typically requires each processor to contain its own individual executive that must be capable of communicating with all the other executives in the total system. This, in turn, will require that each individual executive provide a task handling capability where the tasks resident in various processors can communicate with each other, and, in case of local software or hardware errors, diagnostic capabilities exist to localize "bugs". This is not to say that diagnostic or error checking software is not needed or used in large centralized, single processor systems; however, the diagnostic software development for a distributed systems is usually more difficult and costly.

A distributed processing system is also more dependent on communication technology, particularly where the computers are widely dispersed and the peak traffic demands between the computers are high. The design and development of a unique distributed processor system may require expertise both in hardware and software areas. The advantages and disadvantages of the distributed multiprocessing systems are given in the Table 13 . [Ref. 2]

TABLE 13

MULTIPROCESSOR SYSTEMS ADVANTAGES&DISADVANTAGES

| ADVANTAGES | DISADVANTAGES |
|---|---|
| Increased reliability | Increased H/S complexity |
| Increased survivability | Difficult system testing |
| Increased processing power | Hard failure diagnosis |
| Increased responsiveness | More communications |
| Increased modularity | Depend on com.technology |
| System expandability | Unique expertise needed |

## A. MULTITRANSPUTER CONCEPT

The system performance has increased regularly by a factor of ten each decade in the past as seen in Figure 4.1.

This improvement has been achieved by advances in circuit technology and by increasingly complex systems. VLSI (Very Large Scale Integration) technology offers the potential of much greater circuit complexity for the future but only modest increases in circuit performance.

The economics of uniprocessor systems are based on the historical perspective that processing is expensive in

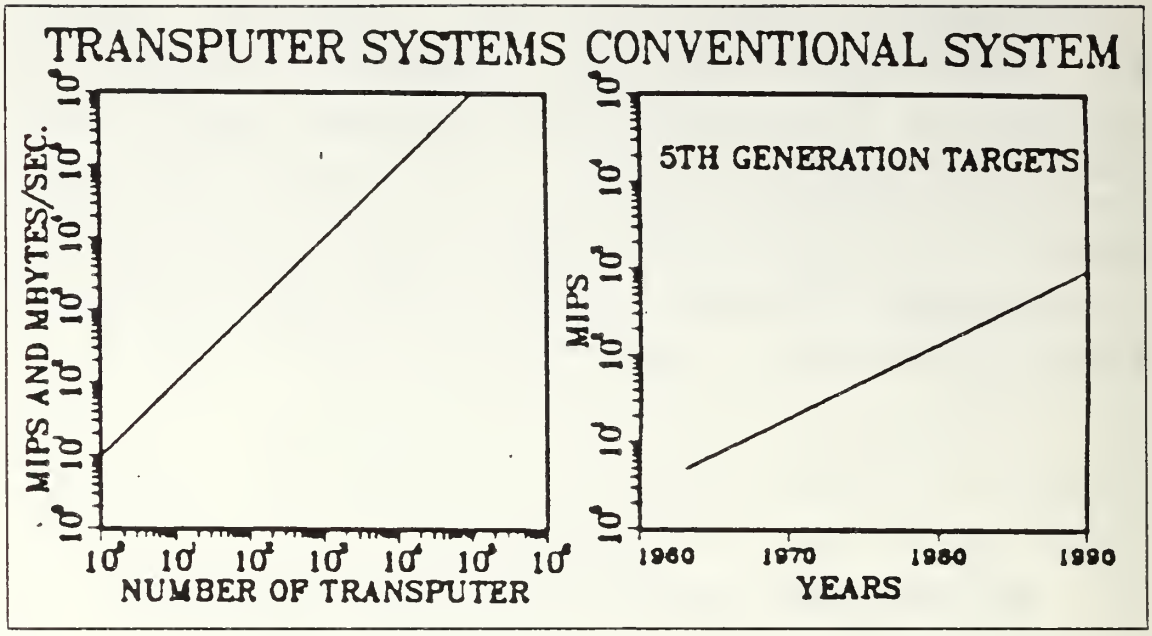


Figure 4.1 Throughputs of the Developing Technology

comparison with the memory. This has led to the Von Neuman bottleneck problem where a single processor is connected to vast amounts of memory. The economics of the VLSI are different. Now, a single wafer of silicon can contain 2 megabytes of memory or 256 conventional microprocessors. To exploit this potential, it will be necessary to build systems with a much higher degree of concurrency than is currently possible.

The transputer is designed as a programmable component for multiprocessing tasks to implement such systems. System architecture is optimized to execute OCCAM, a concurrent programming language. This software sees the system as a collection of concurrent processes that communicate with each other and with peripherals through channels. The same OCCAM program a transputer network executes can run unchanged by a smaller network or a single transputer. The sending transputer transmits messages as a sequence of

bytes, then awaits an acknowledgement. This signifies that the receiving transputer is ready to accept another byte. Transmission is continuous because the receiving transputer acknowledges as soon as it starts to receive a data byte. Moreover, this asynchronous protocol guarantees reliable transmission despite sending or receiving delays. [Ref. 9]

According to the Japanese, an intelligent interaction between people and computers can be achieved with computers which perform a thousand times faster than present day systems. This will only be possible using concurrency, and the transputer has been designed to make such fifth generation systems a possibility. [Ref. 10]

Transputers have special instructions to schedule concurrent processes and to provide communications between them. The transputer does 5 or more MIPS (Mega Instructions Per Second) even when not used in parallel. Hardware supports the parallel-processing language, while memory-intensive architecture speeds execution. [Ref. 11]

A concurrent system is first and foremost a multiprocessor system. The term 'concurrent' is also used in the context of multi-tasking systems; such systems are better described as pseudo-concurrent. Concurrent systems are likely to be no easier to design and implement than non-concurrent systems. Even in such sequential systems, the advantage of design and implementation using a high level language rather than machine-level programming is well recognized; it seems therefore that machines for concurrent systems should encompass the abilities of excellent high level language machines on top of any unique concurrency aspects. [Ref. 12]

Arrays, pipelines and loops of transputers can be used to provide greatly increased performance by exploiting the concurrency inherent in many applications. Two examples which require high performance are signal processing and

database searching. Networks of transputers can provide the performance needed for both applications. Signal processing, such as the Fast Fourier Transform algorithm, maps easily onto a pipeline. The pipeline can accept the input samples at up to 100 KHz., which more than covers the full audio spectrum. A 64 point FFT requires six transputers in the pipeline, a 256 point FFT requires eight and 1024 point FFT requires ten transputers. A pair of pipelines, interlinked at each stage, is able to accept input samples at up to 200 KHz. Higher frequencies can be handled by using more transputers in parallel. [Ref. 6]

An array or a pipeline can also be used to do searching. Provided that the search requests can diffuse through the network and the answers converge, the shape of the network does not matter; it can even contain faulty devices. The full internal memory of each transputer can be searched 1000 times per second. With external memory attached to each transputer, the search rate is slower, but 64 Kbytes per transputer can be searched at least 30 times per second.

If we look at other applications, such as image processing, finite element analysis, matrix manipulation, telephone switching systems, fault tolerant systems and artificial intelligence naturally lend themselves to arrays, loops or networks of transputers.

## B. MULTIPROCESSOR INTERCONNECT STRUCTURES

The traditional approaches to interconnect computers is based on the use of either serial or parallel links. For tightly coupled systems (shared memory) where maximum distances between transmitters and receivers are in the tens of meters range, parallel cables are typically used with 8,16, or 32 bits for data and an equal or perhaps larger number of bits for parity check and control lines.

Several basic design alternatives are available for developing a multiprocessor interconnect structure [Ref. 2]. Information between computers can be transferred from source to destination either directly or indirectly. One or more switching entities may be employed if an indirect transfer strategy is employed. This intervening switching entity may perform and address transformation or route the message onto one of a number of alternative output paths. Examples of systems based on indirect transfer are loops, buses, or star configurations, or packet-switched systems. The major difference between direct and indirect transfer strategy lies in the distribution of message transfer "intelligence". Indirect transfer methods require more complex communications capability but also increase the fault tolerance of a system. Indirect transfer methods are based on either centralized or decentralized routing of messages.

Another design alternative exists in terms of selecting the message transfer path between computers. It may be dedicated as in the case of the loop, star, or completely interconnected system or shared as in the case of bus, packet-switched, or shared memory systems. It may also be a combination of both as in hierarchical systems, where the computer at the top of the pyramid receives messages from several computers, whereas computers at the bottom of the hierarchy have a single path to the computer "above" it.

Generally, a system based on a dedicated path structure is more fault tolerant than a system using shared paths. If a path that is accessible from more than two points fails, no alternative way exists to transfer data between computers in the system. However, systems with redundant paths can be used to minimize the effects of single-point failures on the total system.

Cost of various interconnect schemes depends on whether a system can be developed using off-the-shelf hardware

and/or software or whether the design must be performed "from ground up". Cost is also related to the number of processors to be used in the system, the amount of memory required in each node of the system, and the bandwidth of communications links between computers. The cost of completely interconnected systems tends to be high compared with loop-and bus-based systems. Throughput capacity is as much a function of interconnect structure as it is of link technology. Use of a twisted pair of wires limits data rates to a few megabits per second over a distance of a few thousand feet. Even at this distance, problems are encountered with too many drops if we are dealing with a bus-type system. The bandwidth can, however, be increased using parallel lines, coaxial cable, or fiber optics links. It is, of course, possible to use dial-up or leased telephone lines, but that limits the maximum bandwidth typically to a range of 4800 to 50,000 bit/s. Higher bandwidths are also possible with the use of microwave or satellite links, but this will today have a profound impact on total system cost. [Ref. 2]

Some generalized observations can be made regarding each type of architecture in the areas of cost; modularity, flexibility; reliability, availability, fault tolerance; performance, throughput; ease of development, "off-the-shelfness"; and form factor (design attributes). Depending on how the various design attributes are weighted (which is application dependent), one or more desirable interconnect structures can be selected. The various types of groups of interconnect technology methods are: Complete Interconnection, Packet Switched Network, Regular Network, Irregular Network, Hierarchy, Loop or Ring, Global Bus, Star, Loop with Switch, Bus Window, Bus with Switch, and Shared Memory.

The choice among the architectures is determined by the number of sensors in the system, their physical location, and the amount of data collected by the sensors for transmission to the system processors. Where the number of sensors is extremely large and data rates are very high. The vulnerability to communications path failures can be mitigated using redundant paths. The problems inherent in any architecture are interprogram communication and database considerations, potential deadlock, and error recovery. [Ref. 2]

Loop technology will be explained in details in the next section.

## C. LOOP COMMUNICATION SYSTEM

A Loop multiprocessor system can be defined as a system which consists of a high-speed, unidirectional, digital communication channel (e.g., twisted-wire pair or a fiber optics link) which is arranged as a closed loop or ring. Nodes such as mini or microcomputers, terminals, or peripherals can be attached to the loop channel by a hardware device known as a loop or ring interface. [Ref. 2]

The growing requirements of local communication systems, such as in-house telephone systems, and the introduction of new facilities (alarms, controls, data services, paging) have led to a search for new network concepts. Rather than having several special-purpose networks it would be desirable to provide one single universal system for transmitting and switching of all types of information. Such an integrated system, however, should not depend on complex and expensive central switching equipment. An example of a loop with the various subscriber stations connected at arbitrary points is shown in Figure 4.2[1] the signals are transmitted

[1]Reproduced by permission.

in one direction only. Digital source signals (or sampled and quantized analog signals) offered by the subscribers are parceled and transmitted in blocks to their destination. Assignment of message blocks to subscriber stations is done by using address coding. [Ref. 13]

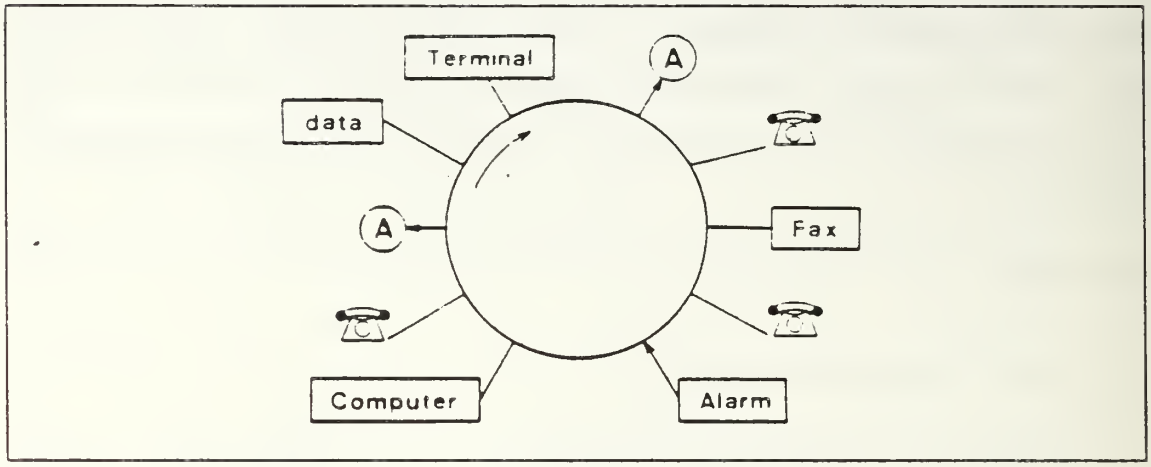


Figure 4.2 Loop Configuration

To send a message from one node to any other, the message is entered on the ring. It will then travel around the ring until it either reaches the node addressed or returns to the transmitting node. In some systems, the originating node removes the message, whereas in others the destination node removes it. In the former case, the originating node can compare the original message with the message which has circulated around the loop, thereby also performing an error check on it. A bit is usually set in a predetermined bit position in the message by the destination node, to signal the transmitting node of message receipt. In the latter case, the destination node removes the message and usually performs the error checking on the message. This approach obviously also reduces the traffic load on the loop.

In a loop system, message transmission takes place in the form of address blocks of data called frames or slots. The local loop interface forms a frame, giving the address of the destination interface, and transmits the frame onto the loop. Each loop interface downstream of the transmitter receives this frame, checks it destination address, and immediately retransmits it back onto the loop if the proper destination for the frame has not been reached. When a receiving loop interface recognizes its own address as the destination of an incoming frame, it removes the frame from the loop and delivers the message to the local attached minicomputer. Digital transmission on the ring is time division multiplexed. The channel capacity of the ring is multiplexed into a series of time slots.

Loop architecture, especially applied to data acquisition, is attractive from a modular point of view. A sensor may be placed anywhere on the loop with a simple interface and may, if necessary, communicate with any node connected to the loop. Since messages are passed from node to node successively, the failure of a single node or path between two nodes can bring down the entire ring. Thus, for unidirectional, active repeater loops, the failure effect and failure reconfiguration attributes are poor. Loop systems are, however, available to improve system fault tolerance (with passive repeaters or bypass relays).

For two-way communications, two channels must be used, but in loop type architectures a single channel is sufficient for communications. This system will obviously fail if any one of the links or nodes fail. The reliability of the loop can, however, be improved using the coupler. Other more reliable loop architectures can also be implemented using fiber optics. [Ref. 2]

The following section will present different types of loops.

## D. TYPES OF LOOPS

Categorization of the loop configurations is based on the type of message-transmission mechanism employed. There are three main types which will be introduced in the following subsections: the Newhall-type, Pierce-type, and Delay Insertion type. [Ref. 2]

### 1. Newhall Type Loop

A control token or character is passed around the loop in a round-robin fashion, from loop interface to loop interface. The interface currently in possession of a token is allowed to transmit messages of arbitrary length onto the loop; the other interfaces are allowed only to receive during this time. The control token is passed to the next node downstream allowing the node to transmit when a transmission is completed. Only one transmitter can be active at any one time, therefore an interface will never experience interference during the transmission of a message. An interface must always wait for the control token to be passed to it, even when it is ready to transmit a message. A Newhall loop provides for variable length message transmission, but it does not allow concurrent use of the loop channel by two or more transmitter interfaces.

### 2. Pierce Type Loop

The communication space on the loop is divided into an integer number of fixed-size slots into which message packets can be stored. It might be considered of as a circular track, with box cars end to end, where some may be full and others are empty. The control of a Pierce type loop is centralized. Each slot contains a bit that indicates whether it is filled with a packet or empty. So, all a transmitter needs to do is to divide each message into

packets, to wait for empty slots to pass by, and when this occurs, to fill them with packets. Because of the slots are fixed size, user messages are blocked into fixed-size packets, prior to being multiplexed onto the line. Various multiplexing techniques can be used.

3. Delay Insertion Type Loop

This loop is superior in overall performance. Each ring interface has a complete set of control capabilities. Delay Insertion Loop has been chosen for our implementation and it will be explained in detail in the next chapter.

## E. LOOP ANALYSIS

### 1. Why Loop ?

A very attractive configuration in multiprocessor systems is a loop, because of its remarkable advantages [Ref. 2]. These advantages are:

a) Only one path for the message to follow in reaching its destination; no message routing problem in system.
b) No transmitter needs to know the location of its receiver.
c) Broadcast message transmission is so easy to achieve, therefore every node can pick up the message.
d) Connections can be established very quickly and easily (important for traffic with short message duration);in several multiprocessor systems based on intermittent inquiry-response nature, messages are usually so short (credit-card verification, electronic fund transfer, goods ordering, information retrieval applications).
e) Digital data transmission eliminates the need for modems and data conversion.
f) Low initial capital investment in loop configurations (cost proportional to number of users of interfaces).
g) A loop configuration provides a very high throughput

(nodal interfaces and not processors are used to relay messages; more messages can be in transmission at the same time).

h) Loops are easy to implement with distributed switching mechanisms without the need for any sophisticated common control (because each loop interface can provide its own bus arbitration and synchronization).

The primary advantage of a loop system is its relatively low cost and high modularity. Node processor failures can be masked by load sharing among the remaining processors if task addresses are kept in tables in each node and checked by the communications software in each computer. The loop approach is particularly attractive when wide-band coaxial or fiber optics buses are used to interconnect the nodes.

A loop is very vulnerable to failures of the interfaces because of its serial organization. Reliability is one major disadvantage, but it can be increased with different methods.

2. Performance of Loop

Some simulation studies on the Distributed Loop Computer Network (DLCN) showed that for low channel utilization the performance of the Newhall loop closely approaches that of the DLCN Delay Insertion Loop. As the traffic level increases, the comparative attractiveness of the Newhall loop diminishes.

The Pierce loop approach is less attractive than either the Delay Insertion Loop or the Newhall Loop, at low levels of line utilization, because a message always has a mean wait time of half a packet interval and must then be transmitted in several packets. At higher traffic levels, the performance of the Pierce loop is better than that of the Newhall loop because of the packet mechanism permits for

two or more concurrently active transmitters. The Pierce loop presents optimal performance characteristics if the message is the same size as the packet. Generally, this does not occur in a real multiprocessor system environment.

A Delay Insertion Loop is more efficient than either of the other two types since every message is composed of message units and the main advantage is that short messages have a short delay time even under heavy loads. The average transmission time on a loop is independent of traffic load for Pierce and Newhall loops. But, the mean transmission time increases significantly with higher traffic loads in the delay insertion loop. The Delay Insertion Loop message transmission technique is superior where queuing delays for messages entering the loop are short.

3. Reliability of Loop

Vulnerability to errors is the major drawback of a loop. Transmission errors can affect the proper functioning of loop organization. A distortion of the receiver address will either result in a packet being delivered to the wrong destination or, if a mutilated address is not being "handled" by the system, a "lost packet " will keep circulating around the loop. Several message-transmission schemes for loop-based systems use a central monitor to check the loop and remove packets that have circled the loop more than once without being received by any of the nodes. Schemes exist where each interface acts as a loop monitor.

Loop interface failures can cause either a loss of access to the loop or a breakdown in loop operation because of the serial nature of a loop. This problem can be solved with electrical relay circuitry and also an additional protection can be provided by opto-isolators.

The reliability of loop configurations can be increased by providing a standby loop that parallels the

main loop. There are two ways, By-Pass and Self-Heal, in which a standby loop can be used in multiprocessor system design.

In the By-Pass technique, traffic can be routed around any number of malfunctioning interfaces, thereby maintaining the connectivity of the loop. The major shortcoming of this technique is the effect of a failure on a reconfiguration unit.

By using Self-Heal technique based on bidirectional double-loop structure, complete connectivity can be maintained when any number of adjacent terminals or reconfiguration units fail. When two nonadjacent nodes or reconfiguration units fail, the sections of the loop on either side of the failure are isolated. This method is highly reliable where a limited number of devices are attached to the loop.

New multiplex systems, redundant communication loops, hierarchical multiloop systems, increasing the stages in a multi-stage loop, and new switch configurations can also increase reliability of the loop.

## F. SYSTEM CONFIGURATIONS WITH THE TRANSPUTER

As seen in Chapter II, the T424 transputer provides four communication channels to use in a system configuration. So, possible system configurations for the transputer may be one of the following structures.

### 1. Matrix Structure

Each computing element in this two-dimensional structure is connected to each other with one channel (If a square matrix is desired for symmetrical structure, the number of computing elements will be 4, 9, 16, 25, 36 and so on). It provides a very large number of communication rings,

and multiple communication paths but there is no further channel redundancy between two computing elements (Figure 4.3).

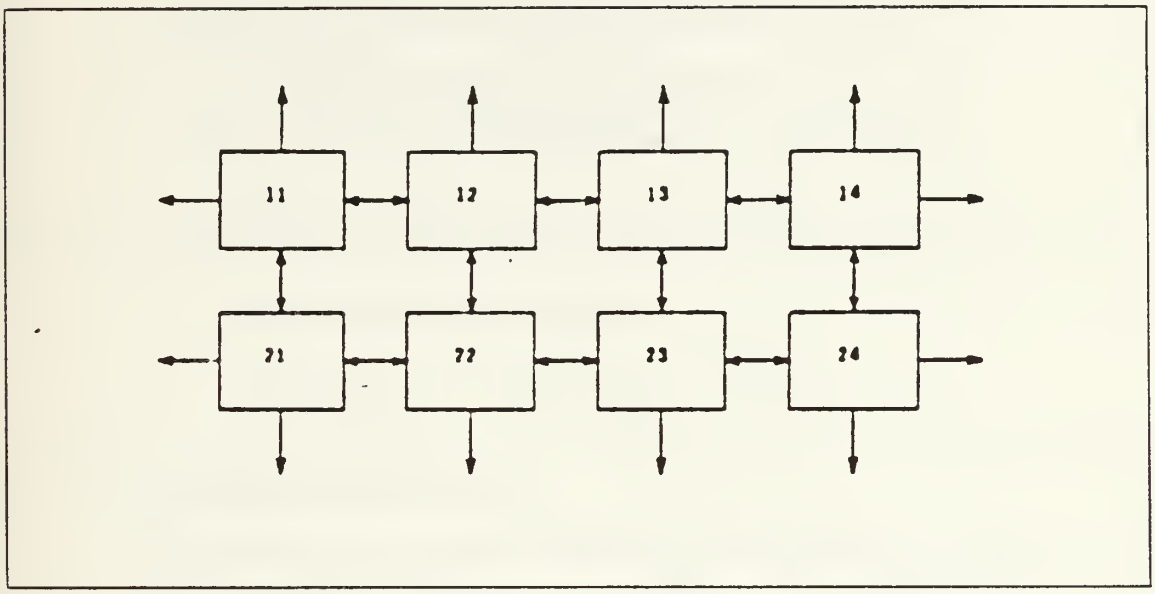


Figure 4.3 Matrix Structure

2. Tetragonal 3-D Structure

In this structure, each computing element is connected to three elements and they build a new computing group which still has four available communication channels for other computing group connections (Figure 4.4).

3. Loop/Ring Structure

Each computing element in this structure is connected to its two neighbors with two channels each. It provides redundancy for communication channels between two computing elements (Figure 4.5).

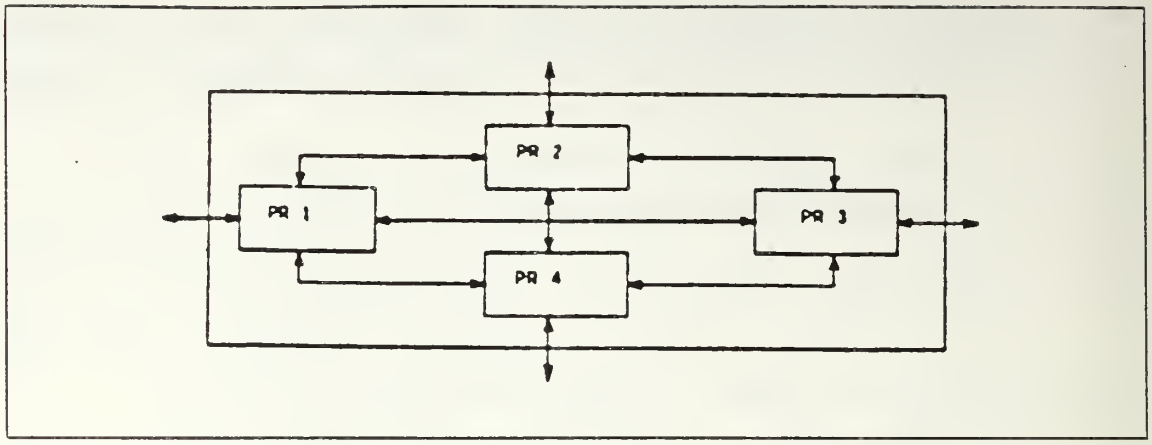


Figure 4.4 Tetragonal 3-D Structure

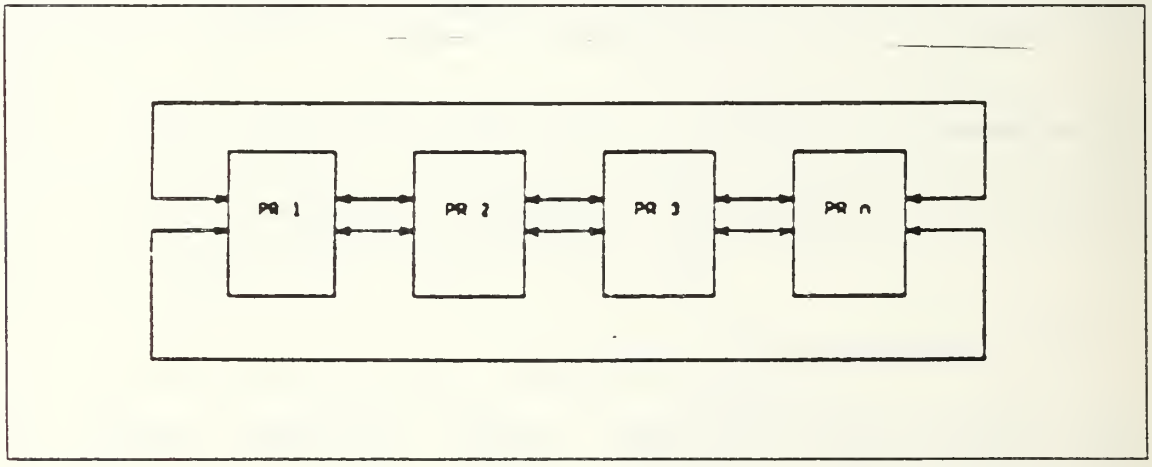


Figure 4.5 Loop/Ring Structure

4. Butterfly Structure

As a special implementation of a ring structure, a butterfly structure is a good solution for the Fast Fourier Transformation or similar engineering applications (Figure 4.6).

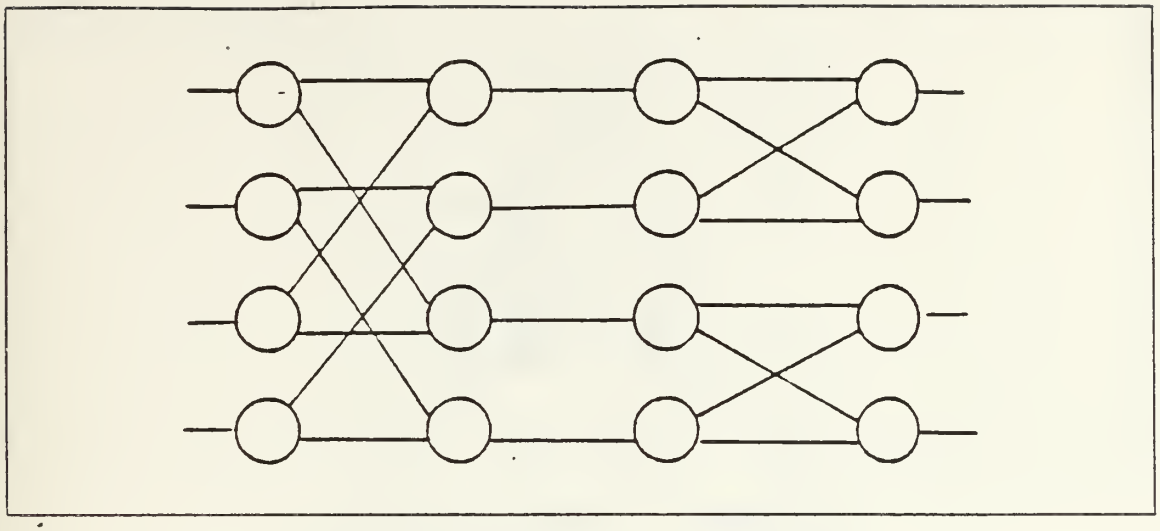

Figure 4.6 Butterfly Structure

5. The Other Structures

There is no limit to the size, function, or shape of a network of multitransputer systems. The transputers can be thought of as building blocks like bricks since they can be built into systems of arbitrary size, function, or shape. Therefore, the system overall performance is a function of the number of transputers. [Ref. 10]

The other possible structures can be a functionally distributed network (Figure 4.7), a toroidally connected array (Figure 4.8), a complete loop regular array (Figure 4.9), and a bigger transputer built from four big transputers (from Tetragonal 3-D Structure) (Figure 4.10).

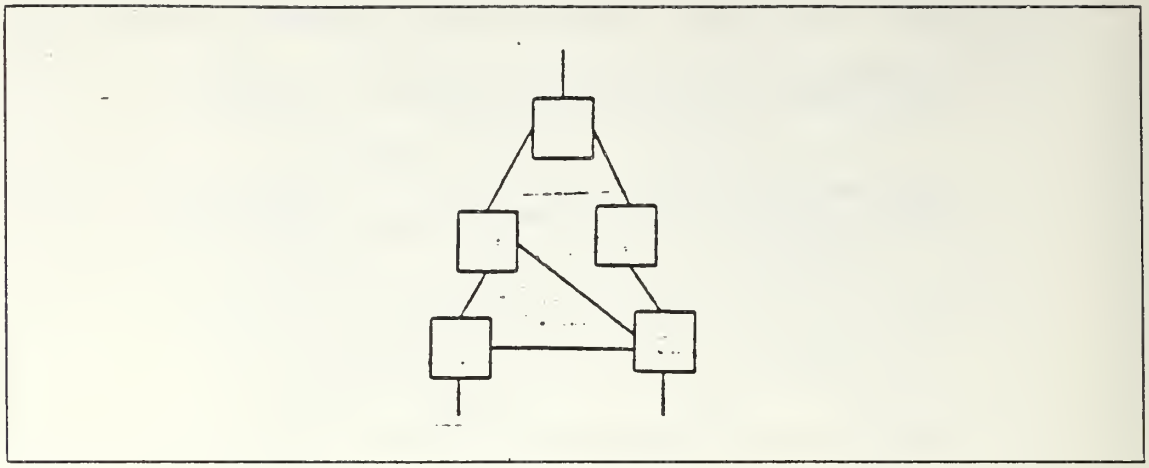

Figure 4.7 A Random Network

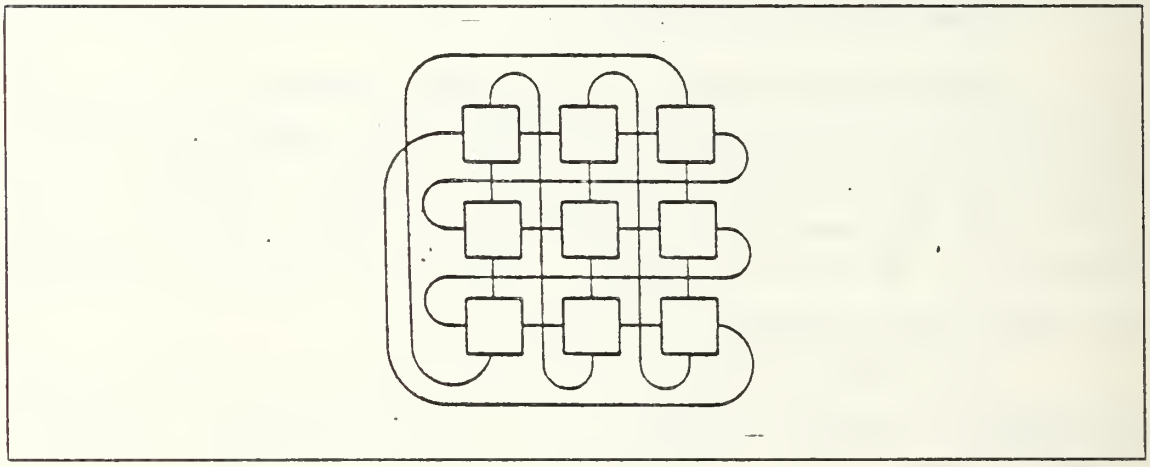

Figure 4.8 A Toroidal Array

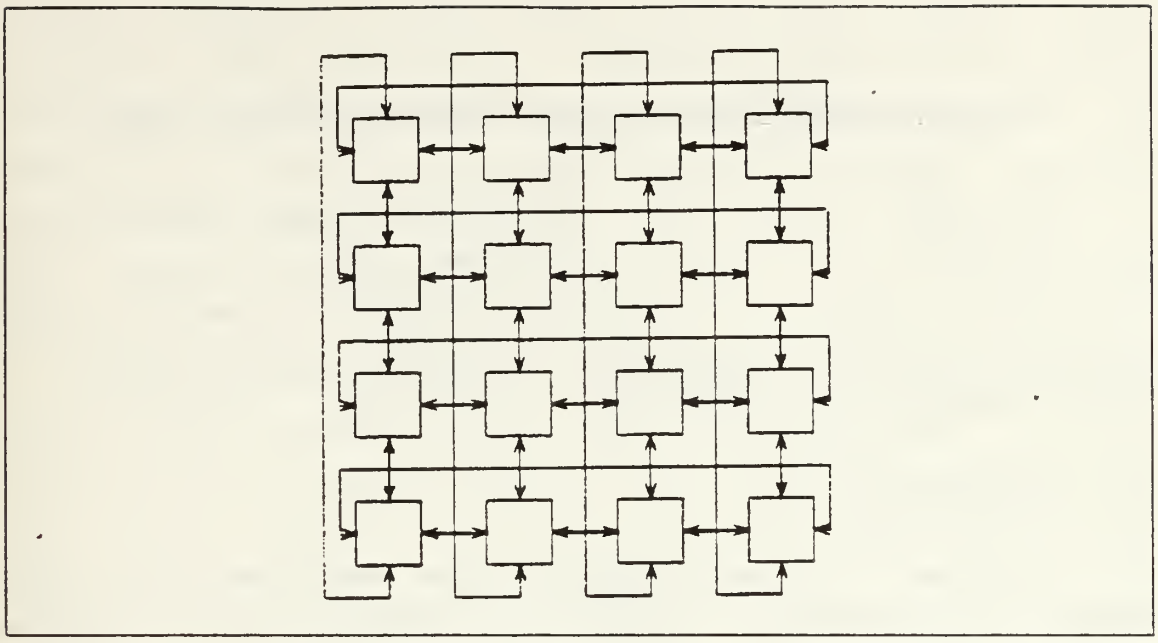

Figure 4.9 A Complete Loop Regular Array

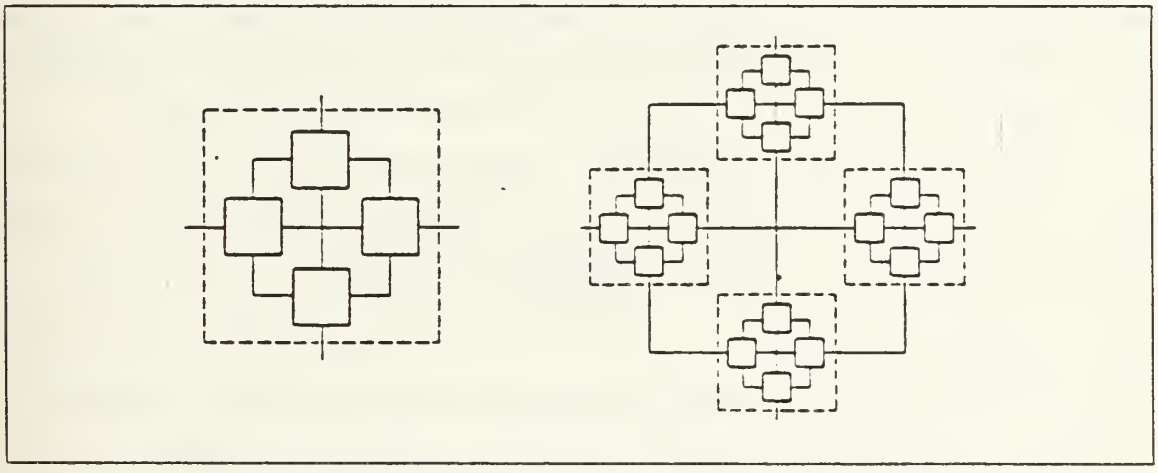

Figure 4.10 A Bigger System Built from Four Big Ones

## V. DELAY INSERTION TYPE LOOP COMMUNICATION SYSTEM

The Delay Insertion Type Loop has a great importance in the concept of a local loop communication systems with distributed control. Various performance studies have shown that its overall performance is superior to Newhall and Pierce loop types. [Ref. 2]

### A. INTRODUCTION

The delay-insertion technique was simultaneously developed by E.R.Hafner, Z.Nenedal, M.Tschanz for telephone switching purposes and by researchers at Ohio State University for the Distributed Loop Computer Network (DLCN) system. The main difference between the two is that variable rather than fixed messages are used in the DLCN scheme. The general operation of Delay Insertion Type Loops is described utilizing the loop communication system developed by Hafner. He refers to this scheme as "loop extension strategy".

Digital signals offered by the subscribers are parceled and transmitted in blocks to their destination using address-coding. Access of the terminals to the loop is gained by switching a delay network (shift register, containing the message block to be transmitted) into the loop line. This strategy guarantees that every station is able to transmit a message at any time regardless of the traffic conditions in the loop. A laboratory model for line frequencies of about 10 megabits/sec is presently being built. It allows data and telephone connections including such features as call back, call transfer and forwarding loop transmission to be realized by means of a three-core symmetrical cable which carries the data sequences and the

timing information separately. This allows the use of an extremely simple regenerator. Possible applications include integrated in-house communication systems and control and supervisory systems for manufacturing processes, railway stations and trains. [Ref. 2]

## B. LOOP ORGANIZATION AND OPERATION

Each subscriber station is equipped for extracting and introducing information from or into the loop. It also contains all the logic functions necessary to control these connections; there is no central exchange. Depending on the task of a station, its structure may be more or less complex, e.g., a receive-only station for one single command will need nothing but a decoder for a fixed address and the coupling element to the line. The basic diagram of a more general station that can transmit information as well as receive is shown in Figure 5.1.[2] [Ref. 13]

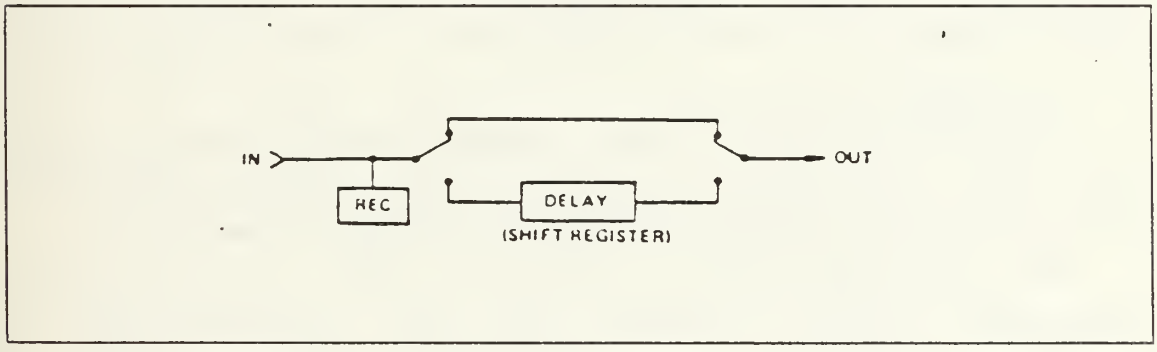


Figure 5.1 Basic Function of Delay Insertion Loop

It contains a delay element, preferably a shift register, which can be switched into the loop line, and a receiver that is permanently connected to the line. The

[2]Reproduced by permission.

delay of the shift register is equal to the length of the message block to be transmitted by this particular station. In the idle state (passive source) the shift register is shunted and the incoming bits or blocks pass on to the outgoing line without significant delay. If a message is to be transmitted, however, the delay element is switched into the line which results in an extension of the loop. The gap generated in the bit stream permits insertion of the message block. This is done automatically by assembling the message packet in the shift register before transmission. The message travels along the loop and is received by every subscriber station. It will only be processed, however, by the station that matches the identification label (address). In the simplest case the message block is taken out of circulation after one entire run at the transmitting station by disconnecting the shift register. This also cancels the loop line extension. [Ref. 13]

For the practical implementation of this basic function, the slightly modified arrangement is used as shown in Figure 5.2.[3] Especially, we will concentrate on this implementation in this thesis.

Each ring interface has a complete set of control capabilities. The detailed diagram shown in Figure 5.3[4] illustrates the basic principle of ring interface operation.

The Receiving Shift Register (RSR) is permanently connected to the incoming line and performs both the receiving and block dropping (removing messages from the loop). There is a second shift register (TSR) for transmitting, i.e., preparing of the message blocks for insertion in the loop from the node processor. Sending and receiving are controlled by a switch (SW) with three positions connecting

---

[3]Reproduced by permission.

[4]Reproduced by permission.

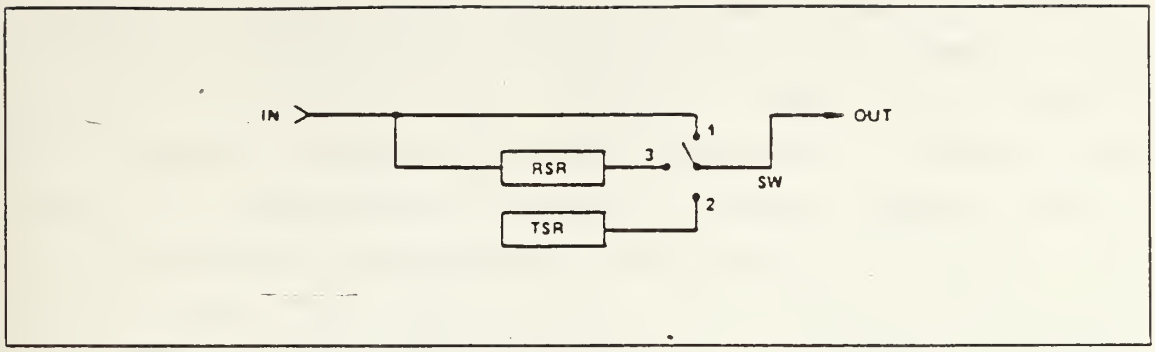


Figure 5.2 Practical Delay Insertion Loop

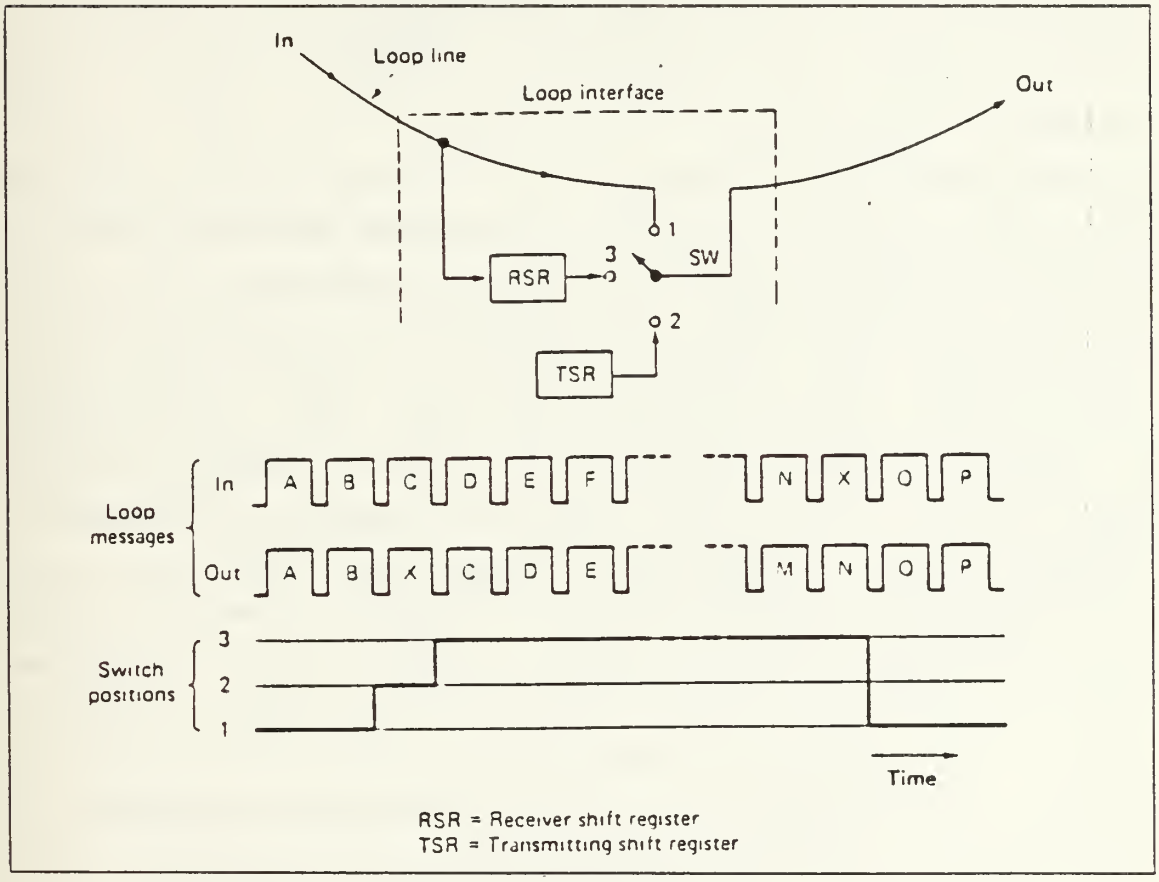


Figure 5.3 Delay Insertion Loop and its Operation

either the output of the TSR, the RSR or the incoming loop line to the outgoing line.

The sequence of events is as follows: When a message is to be transmitted, the three-position switch(SW), initially in Position 1, opens (or cuts) the loop for a well-defined time interval. Since the flow of incoming message blocks (message stream) cannot be stopped without loss of information, the bits arriving during transmission in the RSR have to be stored. Therefore, the arriving bits in the RSR are temporarily buffered and forwarded afterwards. Immediately after a message inserted by the subscriber (node) has left the TSR, which must be of equal length buffer as the RSR, the switch goes to Position 3 and the output is delayed by one message length as shown in Figure 5.3. The node has now entered the loop. It can be called "active" if the switch is in Position 3, as opposed to "passive", if its switch remains in Position 1.

The process of transmitting a block can be initiated only from the passive state. A second message cannot be transmitted before the node has become passive, i.e., left the loop. In other words, for transmitting a second message, the interface must be switched into a passive state (i.e., the node has been disconnected from the loop). This is done by setting the switch from Position 3 to 1, which takes the block in the RSR out of circulation (prevents the block in the RSR from circulating in the loop). The first bit in the following block and the last bit of the preceding one are joined together without leaving a gap. It is clear that switching has to be synchronized in all phases in order to prevent block damaging. Nevertheless every station (node) is entirely autonomous, as it freely determines the moment of transmitting a message (no polling). For leaving the loop, the station must be authorized to take off a particular block. The decision rule is very simple if every station only cancels (removes) its own messages that have circled the loop (Message X in Figure 5.3 is removed before the

switch is returned from Position 3 to 1). This yields a one-to-one correspondence between stations and circulating blocks and allows a free choice of the individual block size. For every new block the entire cycle is repeated as described. The block may contain data, signaling or supervisory information but normally no transmission occurs during idle periods, as opposed to circuit switching systems where channel capacity is reserved even during transmission pauses. [Ref. 13]

Suitable monitoring is of great importance in such a decentralized system. Consider the case where an error is introduced in the address so that neither transmitter nor receiver will recognize its block. If there is no action taken, this block will circulate on forever and finally congest the loop together with other mutilated messages. Meanwhile the originator cannot leave the loop because there is no block he is authorized to remove. This problem can be solved by introducing a monitoring station into the loop at an arbitrary point. In addition to checking single blocks, this station also supervises the entire loop operation and provides clocking and synchronization of the whole system. This monitoring station is a departure from the idea of completely distributed control. Certain specialized functions are more economically performed by common equipment which is at the disposal of all the stations. Examples for telephony are conference call facilities and abbreviated dialing. [Ref. 13]

## C. IMPLEMENTATION OF DELAY INSERTION LOOP

### 1. States of Delay Insertion Loop

We can think of the Delay Insertion Loop Interface as a Finite State Machine with three states corresponding to the three position switch, Figure 5.4.

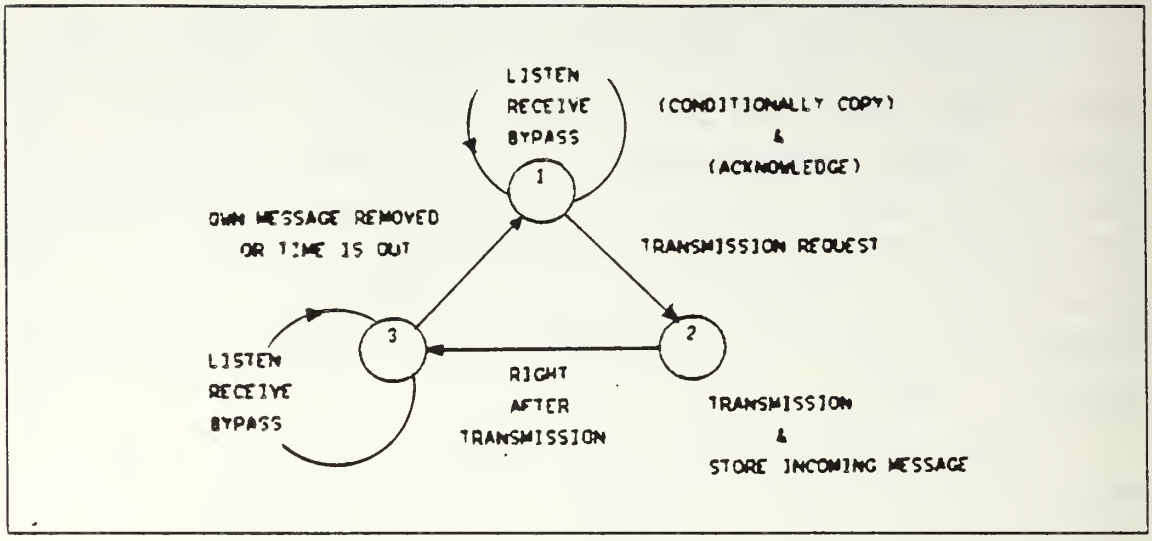


Figure 5.4 States of the Delay Insertion Loop

In State 1 (passive state), the processor listens to inputs from a channel or channels. When no messages come in, then the listening process waits for a well defined time period, and if no messages come in during this period, it is reported as malfunctioning and a new channel is selected. If something does come in, it is checked to see if the message is intended for this processor. If it is, the processor copies and acknowledges the message and passes it on to the next processor. If the message is not for this processor, it is passed without acknowledging and copying.

During the State 1, if a transmission request from user has been generated then a transition to State 2 will be made and the output channel will transmit the message and passes to State 3 immediately after transmission. During the transmission, if there is an incoming message, it is stored. In other words, the receiving process continues.

In State 3, a watchdog timer is set and all incoming messages are checked for originating processor number. If the number is not the same, it means the message is for some

other processor, therefore the message is passed on to the next processor. Otherwise the message is not forwarded and it is checked for acknowledgements. If the acknowledge is not found, an error is reported and the system returns to State 1. If the acknowledge is found, "own" message is removed from the loop, and the system transitions to State 1 and continues its operation.

2. Four Transputer Loop System

The transputer provides four communication channels in a system configuration. Its channel availability and the reliability desire lead us to choose the Two-Loop (ring) or Single-Loop structure because of simplicity for the implementation. In spite of the fact that the system can be affected by loop failure, we will concentrate on the Single Unidirectional Loop (Figure 5.5) because of its less complex implementation.

Now let us take a Single Unidirectional Loop with four transputers as a system, shown in Figure 5.5, and let us apply the Delay Insertion Loop methodology based on the Finite State Machine idea.

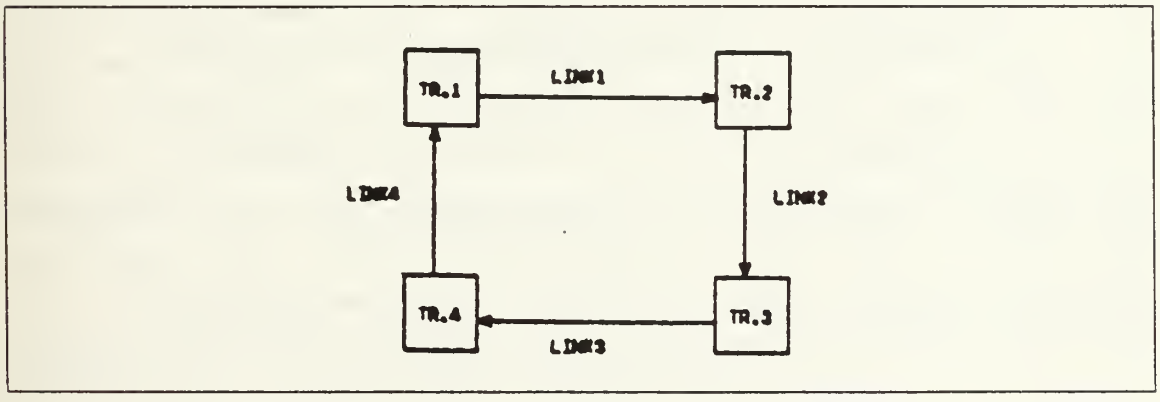


Figure 5.5 Four Transputer Single-Unidirectional Loop

As shown in Figure 5.5, there are four transputers in the system, TR.1, TR.2, TR.3 and TR.4. They are connected with one link (channel) to each other; link1, link2, link3 or link4. Initially every transputer is in State 1. At any instant (say t0), if TR.1 wishes to transmit a message to TR.3, then TR.1 goes to State 2, and the output channel link1 will transmit the message and transition to State 3. TR.2 being in State 1, is listening for messages and the message is received. It is not intended for TR.2, so it will retransmit (just passes) the message through the link2 and remain in State 1. TR.3 is also in State 1, listening to the input channel link2 for the message. It recognizes the message as intended for it, copies, acknowledges and passes it on through link3 staying in State 1. TR.4 is also in State 1, receives and retransmits (passes) the message via link4 and remains in State 1. Finally, after one complete circulation, TR.1 receives the acknowledged message and does not pass it on and transitions to State 1.

All transputers can also transmit a message at the same time being in State 2. Every transputer is initially in State 1. Each transputer has an individual transmission request. Figure 5.6 illustrates the multiple transmission request situation, where horizontal timelines describe what is taking place in each of the four transputers and the adjacent lines describe the activity on the connected links (link 1 of computer 1, L1.1 to link 3 of computer 2, L3.2). The link operation is independent of the processor operation, but the communicating links are completely synchronized in their operation. As shown in Figure 5.6, TR.1 wants to transmit a message (A) to TR.3, TR.2 wants to transmit a message (B) to TR.4, TR.3 wants to transmit a message (C) to TR.1, and TR.4 wants to transmit a message (D) to TR.2 simultaneously, thus each transputer transitions to State 2. After the transmission, they transition to State

3 for simultaneous listening and receiving their own messages. All the operations occur at the same time as explained before. In our implementation in the simulated mode on the VAX 11/780 VMS system, these operations occur concurrently with the uniprocessor switching from one process to another by timeslices.

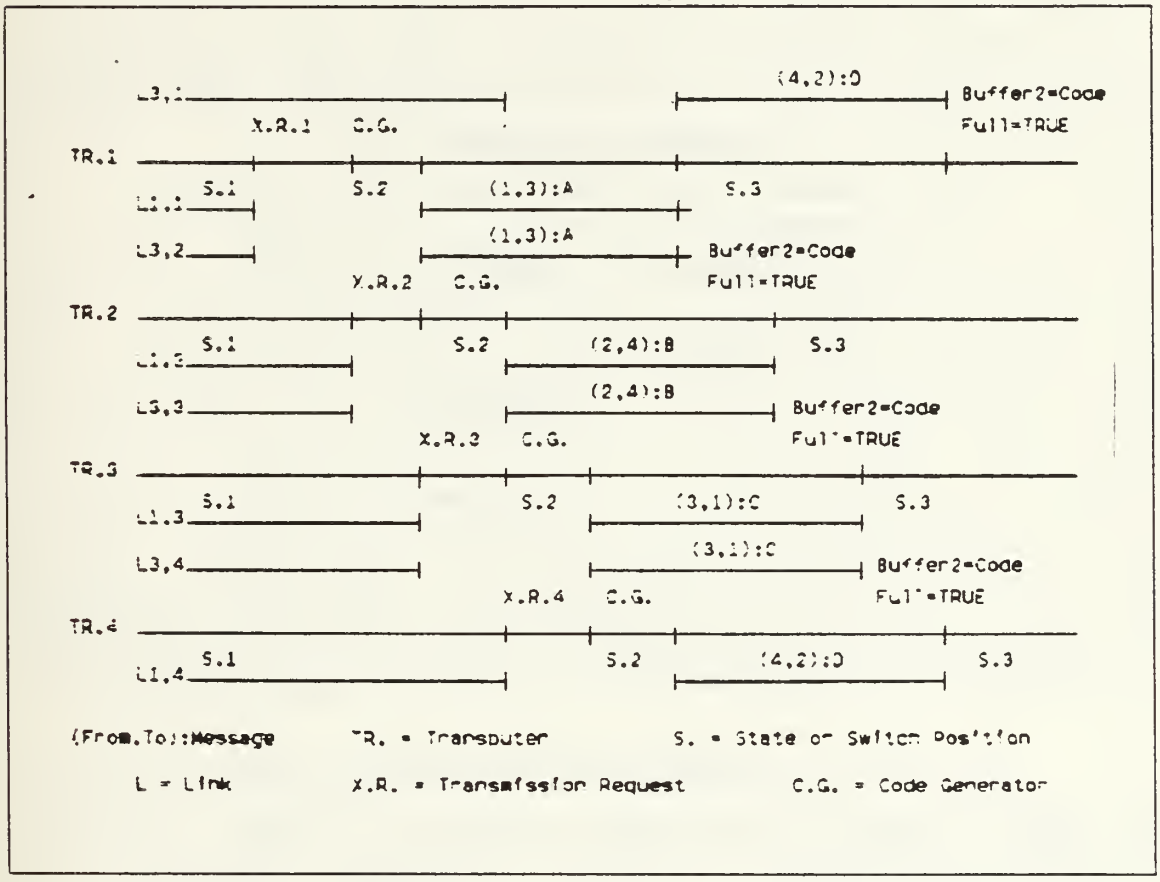


Figure 5.6 Simultaneous Transmission of Four Transputers

Software imlementation of this system will be presented in the following subsection.

## 3. OCCAM Implementation of the System

The Delay Insertion Loop Interface in our implementation, as shown in Figure 5.7, provides inter and intra communication between systems, transputers and processes. It also provides a fault detection (error check) feature during communications, using watchdog timers and acknowledging techniques.

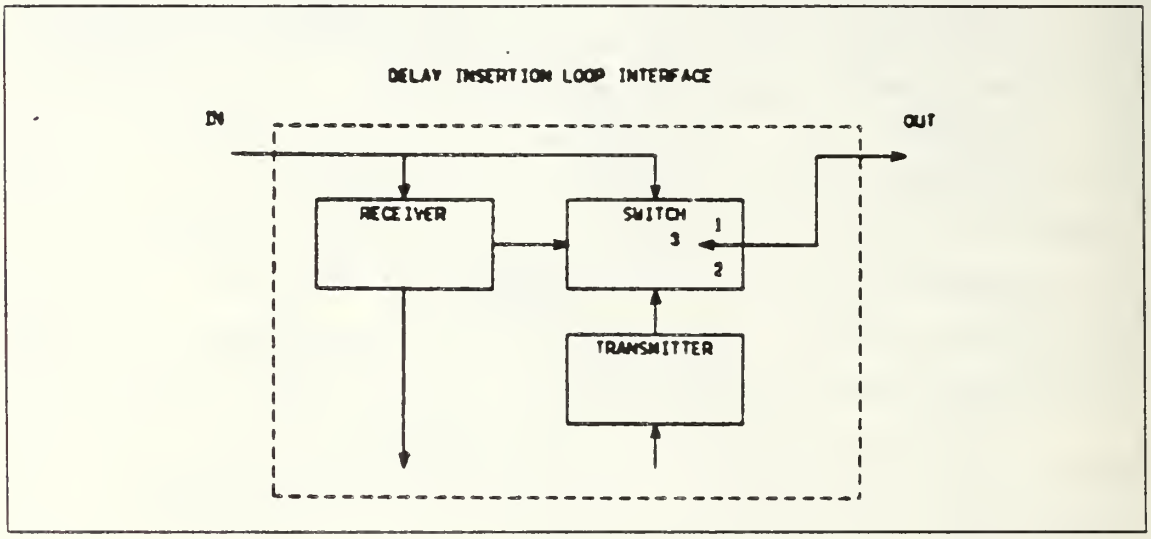


Figure 5.7 Implementation of Delay Insertion Loop

The Delay Insertion Loop Interface Process (Appendix A) accepts inputs from links or software (OCCAM) channels. At proper instances in time, the Delay Insertion Loop Interface Process sends messages through the transputer links or OCCAM channels. Three different communication types are achieved by D.I.Loop.Interface. These are by-pass (from outer transputer to outer transputer), internal disribution (from outer transputer to inner process) and external distribution (from inner process to outer transputer). More briefly, D.I.Loop.Interface listens to link and user channel, determines the communication type, transmits and receives the message through the channel or link.

a. Message Format

A decimally coded message block communication protocol is used to obtain efficiency. For simplicity, there is no data passed in the message block. The message block includes the message type, destination transputer number, source transputer number, process number and message responses (error, receive etc.). This coded message block is accepted in coded form by the loop interface system and is used by the same system to determine what should be done with it. The coded message block has been named as CODE in the implementation. The CODE is a binary 32 bit two's complement word interpreted as a decimal integer in the range -2147483648 to 2147483647. Only positive values are used. The first digit of the ten-digit code represents the message type. This digit is not used in our implementation, but it can be used for priority (0,1,2). The next three two-digit groups are used to show destination transputer number, source transputer number and process number respectively. Each value ranges between zero and 99, so 100 transputer addresses and 100 processes can be used in this system as a maximum. Table 14 shows the CODE and its digits.

b. The Algorithm of the System

D.I.Loop.Interface process (procedure) (Appendix A) has an input channel IN, an output channel OUT and corresponding transputer number TR.NO as formal parameters. In local declarations, TRANSMISSION.REQUEST and SENDER.CHANNEL as local channels, TIMEOUT as a constant (one circulation period), SWITCH.POSITION, CODE, DEST.TR.NO, SOURCE.TR.NO, PR.NO, CLOCK, FULL, BUFFER1 and BUFFER2 are declared as local variables.

There are two processes (procedures) in D.I.Loop.Interface, CODE.GENERATOR and DECODER.

TABLE 14
CODE AND ITS DIGITS

| max. value | 2147483647 | |
|---|---|---|
| code digits | xddsspprrr | x : msg.type(0,1,2)<br>dd : dest.tr.no<br>ss : source tr.no<br>pp : process no<br>rrr: responses |

CODE.GENERATOR process generates the CODE interpreted as 10 decimal digits which includes DEST.TR.NO (Destination Transputer Number), SOURCE.TR.NO (Source Transputer Number) and PR.NO (User Process No to be executed). The DECODER process decodes the CODE by using division and remainder operations, and determines destination, source transputer number and user process number.

The general system is implemented in an infinite loop by using WHILE TRUE. The variable FULL is initialized with FALSE, CLOCK with NOW function and SWITCH.POSITION with 1 (switch is initially at 1). The variable SWITCH.POSITION will simulate the positions of the switch and provide the states. Three states, WHILE switch.position=1, WHILE switch.position=2 and WHILE switch.position=3 are executed sequentially being in SEQ construct.

In State 1 ('WHILE switch.position=1'), three operations or three ALT (alternative) constructs execute in parallel in PAR construct. In the first ALT (this portion represents the receiver), there are two guards, first is an input and the second is a special wait statement. In the first guard, 'in ? code', if there is an incoming message

(CODE) on input channel IN, it is decoded by DECODER, then it is checked for destination address. 'IF dest.tr.no=tr.no' (if message is for us) then the message (CODE) is copied into a buffer (BUFFER1). SENDER.CHANNEL sends user process number (PR.NO) for execution. If message is not for us (here, it is TRUE), OUT channel passes the CODE to the next transputer (By-pass, just passes the message without doing anything). In the second guard, there is a special construct 'WAIT NOW AFTER clock+timeout'. This construct provides a wait function for a period of time (timeout is one complete message block execution time). If time is out (if there is no incoming message and time period expires), then its component process executes, a "TIME IS OUT. NO INCOMING MESSAGE" on the CRT screen. In the second ALT (alternative) construct, there are two guards. The first one is 'full & SKIP', SKIP is always TRUE, if full is TRUE (it means BUFFER2 is full with message), then its component process is executed (here OUT channel sends the message in BUFFER2). The second guard is a special WAIT function as seen above, it provides wait time, if TIMEOUT period expires (time is out), then its component process executes, a "TIME IS OUT" message on the CRT screen. In the third ALT construct, there are again two guards, one is 'transmission.request ? ANY' and the other is special WAIT statement as seen before. If there is a transmission request from the user, then a TRANSMISSION.REQUEST channel will have a signal and then its component process 'switch.position := 2' will be executed, so the system will go to State 2. If time is out (second guard), there is no transmission request from the user, then the wait time period for the request expires and a "TIME IS OUT. NO XMISSION REQUEST" message is provided.

In State 2 ('WHILE switch.position = 2'), one ALT and one SEQ construct are executed in parallel. In ALT

part, one of two guards is 'in ? code'. If there is an incoming message during transmission, its component SEQ process is executed. Incoming message (CODE) is stored in buffer (BUFFER2), then BUFFER2 becomes full with message and variable FULL becomes TRUE. If there is no incoming message during transmission period, wait time expires (second guard in ALT) and then "TIME IS OUT. NO INCOMING MESSAGE" will be displayed on the CRT screen. In SEQ construct inside PAR (this part represents the transmitter), the message CODE is generated by CODE.GENERATOR, and OUT channel transmits the CODE and then SWITCH.POSITION becomes 3 immediately (the system goes to State 3).

In 'WHILE switch.position = 3' (State 3), two alternative constructs execute in parallel inside the PAR construct. In the first ALT, if variable FULL is TRUE, in other words if BUFFER2 is full with message, OUT channel sends the message from BUFFER2. Otherwise a wait time expires while the receiver waits for an incoming message, then "TIME IS OUT" message is displayed on the CRT screen. In the second ALT inside the PAR construct, if channel IN has a CODE (if there is an incoming message), then the message CODE is decoded by DECODER and it is checked for the originating (source) transputer number. 'IF source.tr.no = tr.no' (if the transputer generated the message), then its own message is removed from the loop to prevent the further circulation of the message. The SWITCH.POSITION becomes 1, and the system turns back to State 1. If the message is not its own, OUT channel forwards the message CODE. In the second guard, wait time expires if its own message is not received, then "TIME IS OUT" message is sent to the CRT screen. If a fault tolerance system is designed, it can be activated here for faulty communication.

In the main program part, actual channels link1, link2, link3 and link4 are declared and four

D.I.Loop.Interface processes are executed in parallel inside the PAR construct with the corresponding transputer channels and the transputer numbers as actual parameters. In other words, four transputers are run in parallel.

As seen in our program segment, OCCAM provides very useful and effective tools for concurrent processing. But, in spite of the fact that our program matches completely with OCCAM Reference Manual [Ref. 5], the keyword NOW (local time function, provides present time) is not accepted in its syntax checker and compiler. Therefore, the WAIT function with NOW and AFTER, as a watchdog timer couldn't be implemented in the VAX 11/780 VMS system.

# VI. CONCLUSIONS

## A. SUMMARY

We have constructed a model of a Delay Insertion Loop interface to interconnnect clusters of computers. The Transputer T424, a hardware component which is becoming available soon (December 1985), was used in a simulated mode on the VAX 11/780 to construct the model of the Delay Insertion Loop interface. The programming language OCCAM which allows concurrent processes to be executed in parallel was used to create the model for multiprocessing in a multitransputer system.

The features of OCCAM and the capabilities of the T424 Transputer were presented in detail. The possible structures with the Transputer in multiprocessing environment, the possible multitransputer systems and the ideas of multiprocessing were explained.

The Loop technology was examined and the Delay Insertion Loop was emphasized and some suggested configurations with four and sixteen transputers were made. Using OCCAM and the Transputer, The Delay Insertion Loop Interface was implemented for the Four-Transputer Single-Unidirectional Loop System using the VAX 11/780 VMS system.

## B. RESULTS AND COMMENTS

This thesis work showed us that the Delay Insertion Loop Network Interface with in a multitransputer system is a very attractive alternative and likely to achieve high performance in many present applications. It may become a good candidate for many military real time applications in the future.

The Delay Insertion Loop methodology is not limited to multitransputer systems, it can also be used with any multiprocessor or multiprocessor cluster computer systems.

The single unidirectional loop is a very simple communication system to implement, but it is subject to loop failure. Adding a second loop to the system can be one of the practical methods to increase the reliability.

The concurrency can be used to provide considerable gains in performance in many application areas. The T424 transputer, is a new product with high performance, which allows concurrent processing. It is an improving hardware component in a new phase of the computer technology. To design concurrent systems is difficult even with a relatively simple hardware architecture like the T424 Transputer. However, as understanding of concurrent processing improves, very powerful multitransputer systems can be generated to support real-time processing.

OCCAM, based on a model of concurrency, is a simple language in which to learn to write programs. But the concurrent processing is hard to understand and to implement, especially for inexperienced people. Some expertise is required for the application of the concurrent system. If the number of processors or transputers in a system is increased, the software implementation becomes more complex, and especially the communication between transputers or transputer systems will be more complicated.

In software programming with OCCAM, some execution errors such as Deadlock, Stop and Access Violation can be often encountered until one is familiar with the PAR (parallel), ALT (alternative) constructs, and using channels for input and output. The other constructs are so easier and especially the SEQ (sequential) construct provides statement by statement execution like the other conventional programming languages.

We have worked using the initial version of OCCAM and its compiler which was installed into our VAX 11/780 VMS system. Therefore, we have encountered some difficulties and problems (such as run-time checking, some unacceptable constructs and keywords like NOW, PLACEDPAR and PRIPAR in syntax checking). But new updated versions of the OCCAM may not have such problems.

## C. SUGGESTIONS FOR FOLLOW-ON WORK

This thesis addressed only to the implementation of the Delay Insertion Loop Interface with a four transputer single-unidirectional loop system. Possible continuation of this work may be the implementation with single-bidirectional, bidirectional two loop or the complete loop system with sixteen transputers.

In a bidirectional two loop system structure as shown in Figure 6.1, each transputer is connected to two neighboring transputers with two channels. The message traffic can flow in both directions, but circulating traffic in one direction is less complicated. The system is unaffected by a single loop failure. As seen in Figure 6.1, the loops can be named ODD and EVEN. Always one loop is on duty (active) while the other rests (passive). If a failure occurs while the active one runs, the spare one (which waits idly) takes over its job immediately.

In regular array complete loop system as shown in Figure 6.2, there are sixteen transputers (or clusters of transputers) each one is connected with one link to each other. There are four vertical and four horizontal loops in the system. If one of the link fails, the corresponding loop is canceled and removed from the system without serious effects.

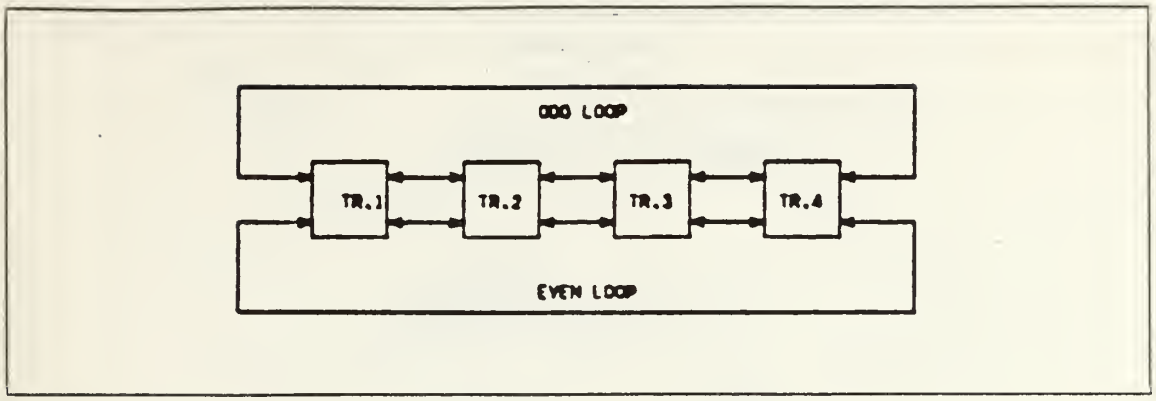


Figure 6.1 Two-Loop System with Four Transputers

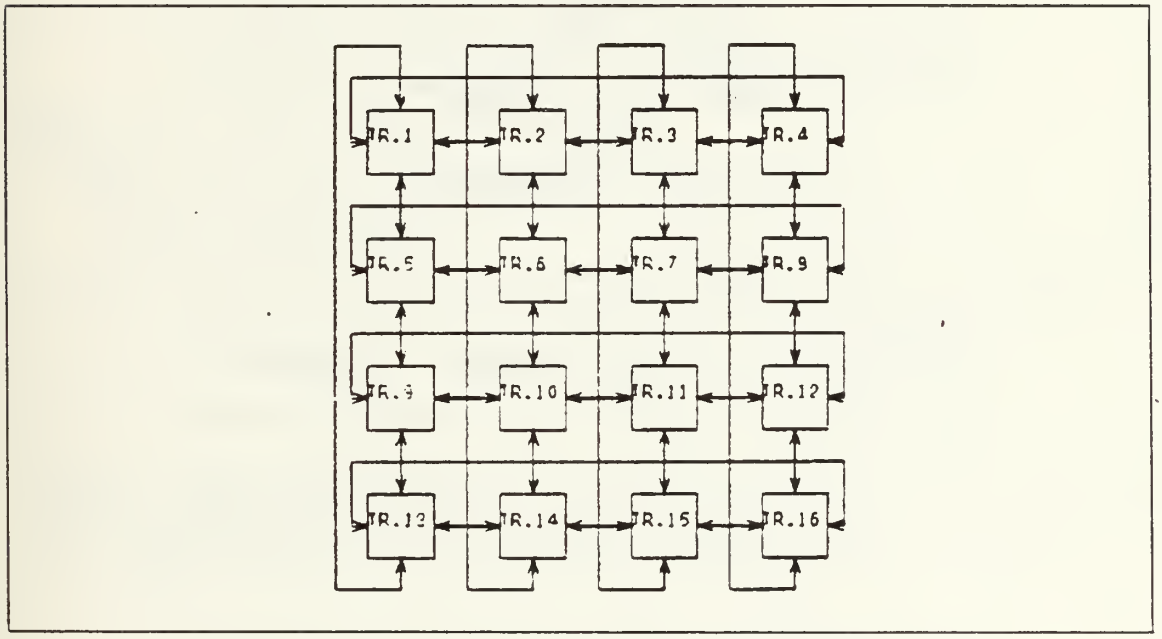


Figure 6.2 16 Transputer Regular Array Complete Loop

The multicluster shared memory system, shown in Figure 6.3 [Ref. 14,], is also suggested for further implementation of Delay Insertion Loop methodology.

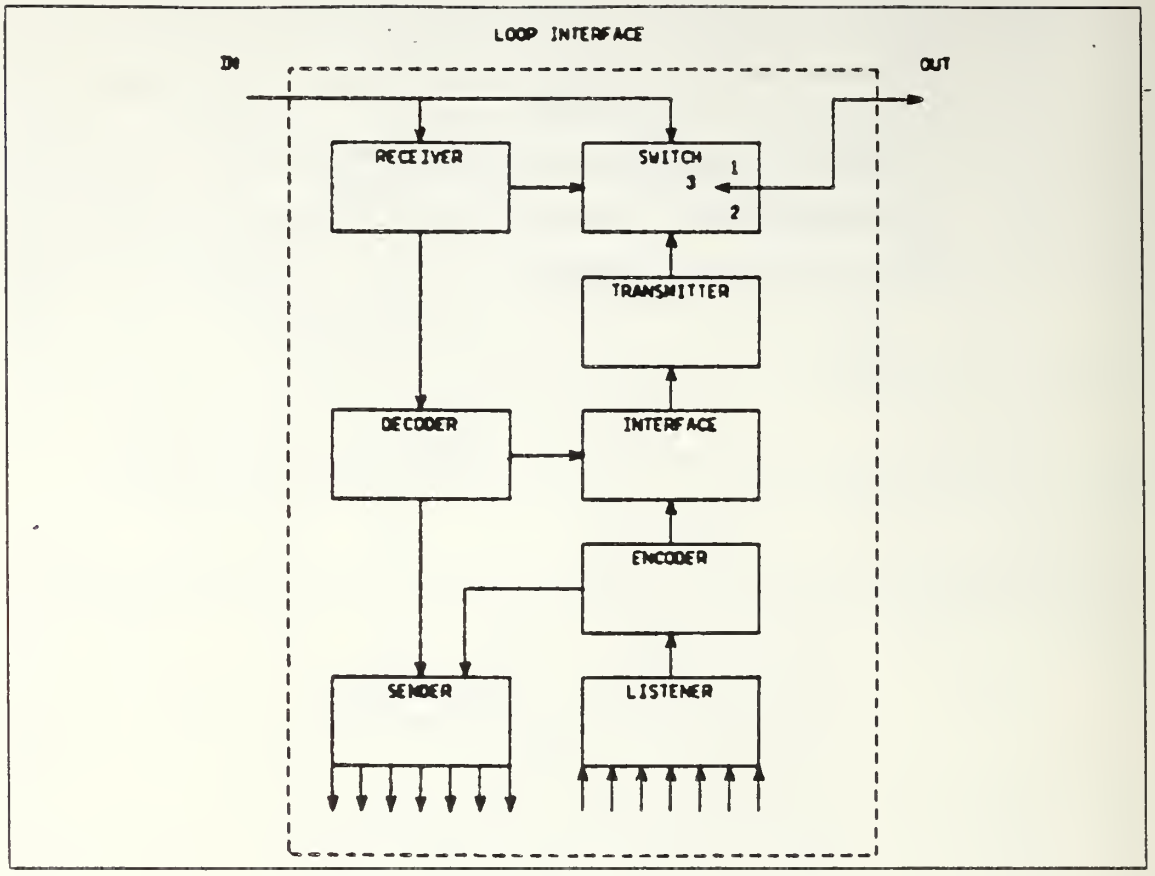


Figure 6.3 Suggested Loop Interface

# APPENDIX A

## DELAY INSERTION LOOP INTERFACE

```
--        ********************************************
--        *                                          *
--        *           GLOBAL DECLARATIONS            *
--        *                                          *
--        ********************************************

CHAN screen   AT 1 :
CHAN keyboard AT 2 :
DEF  end.buffer = -3 :
VAR  char.string [BYTE 512] :


--        ********************************************
--        *                                          *
--        *  WRITING ANY CHARACTER STRING TO CRT     *
--        *                                          *
--        ********************************************

PROC write.screen (VALUE string[]) =
  SEQ
    SEQ i = [ 1 FOR string[BYTE 0] ]
      screen ! string [BYTE i]
    screen ! end.buffer :


--        ********************************************
--        *                                          *
--        *     DELAY INSERTION LOOP INTERFACE       *
--        *                                          *
--        ********************************************

PROC D.I.Loop.Interface ( CHAN in,out, VALUE tr.no ) =
  CHAN transmission.request, sender.channel :
  DEF timeout = 4 :
  VAR switch.position, code, dest.tr.no, source.tr.no,
      pr.no, clock, full, buffer1, buffer2 :


--        ********************************************
--        *                                          *
--        *         CODE GENERATOR PROCESS           *
--        *                                          *
--        ********************************************

  PROC Code.Generator =
    code := (dest.tr.no*10000000) + (source.tr.no*100000)
            + (pr.no*1000) :
```

```
--        ********************************************
--        *                                          *
--        *             DECODER PROCESS              *
--        *                                          *
--        ********************************************


PROC Decoder =
  SEQ
    dest.tr.no    := code/10000000
    source.tr.no  := (code\10000000)/100000
    pr.no         := (code\100000)/1000    :



WHILE TRUE                                        -- infinite loop
  SEQ
    full := FALSE                               -- buffer2 is empty
    clock := NOW                                  -- present time
    switch.position := 1              -- switch is initially at 1



--        ********************************************
--        *                                          *
--        *                STATE 1                   *
--        *                                          *
--        ********************************************


    WHILE switch.position = 1                            -- State 1
      PAR
        ALT
          in ? code                             -- incoming message
            SEQ
              Decoder                        -- decode the message
              IF
                dest.tr.no = tr.no    -- if message is for us
                  SEQ
                    buffer1 := code          -- copy the message
                    sender.channel ! pr.no --send to execute
                    out ! ANY                -- acnowledge signal
                TRUE                -- if the message is not for us
                  out ! code              -- just pass the message
          WAIT NOW AFTER clock + timeout     -- if time is out
            write.screen ("TIME IS OUT.NO INCOMING MESSAGE")
        ALT
          full & SKIP                           -- if buffer2 is full
            out ! buffer2          --send the message in buffer2
          WAIT NOW AFTER clock + timeout     -- if time is out
            write.screen ("TIME IS OUT")
        ALT
          transmission.request ? ANY    -- xmission request ?
            switch.position := 2                -- go to State 2
          WAIT NOW AFTER clock + timeout     -- if time is out
            write.screen ("TIME IS OUT.NO XMISSION REQUEST")
```

```
--        ********************************************
--        *                                          *
--        *                 STATE 2                  *
--        *                                          *
--        ********************************************

    WHILE switch.position = 2                          -- State 2
      PAR
        ALT
          in ? code     -- incoming message during xmission ?
            SEQ         -- if there is an incoming message
              buffer2 := code                  -- store in buffer2
              full := TRUE   -- buffer2 is full with message
          WAIT NOW AFTER clock + timeout    -- if time is out
            write.screen ("TIME IS OUT.NO INCOMING MESSAGE")
        SEQ
          Code.Generator    -- generate message for xmission
          out ! code                       -- transmit the message
          switch.position := 3  -- then turn back to State 3

--        ********************************************
--        *                                          *
--        *                 STATE 3                  *
--        *                                          *
--        ********************************************

    WHILE switch.position = 3                          -- State 3
      PAR
        ALT
          full & SKIP      -- if buffer2 is full with message
            out ! buffer2      -- send the message in buffer2
          WAIT NOW AFTER clock + timeout    -- if time is out
            write.screen ("TIME IS OUT")
        ALT
          in ? code           -- if there is incoming message
            Decoder                        -- decode the message
            IF
              source.tr.no = tr.no    -- if it is own message
                SEQ
                  code := 0  -- remove the message from loop
                  switch.position := 1  -- return to State 1
              TRUE                  -- if it is not own message
                out ! code              -- just pass the message
          WAIT NOW AFTER clock + timeout    -- if time is out
            write.screen("TIME IS OUT")

--        ********************************************
--        *                                          *
--        *               MAIN PROGRAM               *
--        *                                          *
--        ********************************************

CHAN link1, link2, link3, link4 :
PAR
  D.I.Loop.Interface (link4, link1, 1)
  D.I.Loop.Interface (link1, link2, 2)
  D.I.Loop.Interface (link2, link3, 3)
  D.I.Loop.Interface (link3, link4, 4)

--        ********************************************
--        *                                          *
--        *     THE END OF THE OCCAM PROGRAM         *
--        *                                          *
--        ********************************************
```

# LIST OF REFERENCES


1. Madnick, S.E., Donovan, J.J., Operating Sytems, McGraw-Hill, New York, 1974.

2. Weitzman, C., Distributed Micro/MiniComputer Systems, Prentice-Hall, New Jersey, 1980.

3. INMOS Limited, IMS T424 Transputer Reference Manual, 1984.

4. INMOS Limited, IMS T424 Transputer Advance Information, 1984.

5. INMOS Limited, OCCAM Programming Manual, 1983.

6. INMOS Technical Note 5, The Design of Concurrent Systems, 1984.

7. INMOS Technical Note 10, Floating Point Arithmetic with the IMS T424, 1985.

8. INMOS Technical Note 6, OCCAM, 1983.

9. "Integrated Circuits", Computer Design, Decoember 1984.

10. Walker, Paul, "Multiprocessing-The Transputer", Byte, May 1985.

11. Baron, I., Cavill, P., May, D., Wilson, P., Technical Article, Electronics, November 1983.

12. INMOS Technical Note 9, Architectural Decisions in the Design of the IMS T424, 1984.

13. Hafner, E.R., Nenedal, Z., Tschanz, M., "A Digital Loop Communication System", IEEE Transactions on Communications, June 1974.

14. Selcuk, Z., Implementation of a Serial Communication Process for a Fault Tolerant, Real Time, Multi Transputer Operating System, M.S. Thesis, Naval Postgraduate School, Monterey, 1984.

# INITIAL DISTRIBUTION LIST

| No. | | Copies |
|---|---|---|
| 1. | Defense Technical Information Center<br>Cameron Station<br>Alexandria, Virginia 22304-6145 | 2 |
| 2. | Library, Code 0142<br>Naval Postgraduate School<br>Monterey, California 93943-5100 | 2 |
| 3. | Department Chairman, Code 52<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, California 93943-5100 | 1 |
| 4. | Prof. Uno R. Kodres, Code 52 Kr<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, California 93943-5100 | 3 |
| 5. | Prof. Roger Marshall, Code 52 Mi<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, California 93943-5100 | 1 |
| 6. | Daniel Green, Code 20E<br>Naval Surface Weapons Center<br>Dahlgren, Virginia 22449 | 1 |
| 7. | CAPT J. Donegan, USN<br>PMS 400B5<br>Naval Sea System Command<br>Washington, D.C. 20362 | 1 |
| 8. | RCA AEGIS Depository<br>RCA Corporation<br>Government Systems Division<br>Mail Stop 127-327<br>Moorestown, New Jersey 08057 | 1 |
| 9. | Library (Code E33-05)<br>Naval Surface Weapons Center<br>Dahlgren, Virginia 22449 | 1 |
| 10. | Dr. M. J. Gralia<br>Applied Physics Laboratory<br>John Hopkins Road<br>Laurel, Maryland 20707 | 1 |
| 11. | Dana Small<br>Code 8242, NOSC<br>San Diego, California 92152 | 1 |
| 12. | Genelkurmay Baskanligi<br>OBID<br>Bakanliklar, Ankara, Turkey | 1 |
| 13. | K.K.K.ligi<br>Pl. Pren. Bsk. KOKOBI<br>Yucetepe, Ankara, Turkey | 1 |

14. Dz.K.K.ligi 1
Personel Daire Bsk.ligi
Bakanliklar, Ankara, Turkey

15. Dz.Harp Ok.K.ligi 1
Fen Bilimleri Bl. Bsk.ligi
Heybeliada, Istanbul, Turkey

16. Dz.Harp Ok.K.ligi 1
Kutuphanesi
Heybeliada, Istanbul, Turkey

17. Istanbul Teknik Universitesi 1
Kutuphanesi
Istanbul, Turkey

18. Istanbul Universitesi 1
Kutuphanesi
Beyazit, Istanbul, Turkey

19. Bogazici Universitesi 1
Kutuphanesi
Istanbul, Turkey

20. Orta Dogu Teknik Universitesi 1
Kutuphanesi
Ankara, Turkey

21. Hacettepe Universitesi 1
Kutuphanesi
Ankara, Turkey

22. Zafer Selcuk 1
Haznedar Hurmali sk. Gunes apt. 3/4
Bahcelievler, Istanbul, Turkey

23. Gene Allard 1
Box 236
Jefferson, South Dakota 57038

24. Bekir Evin 3
Akdurak mh. Kefken cd. 8/3
Kandira, Kocaeli, Turkey

Thesis
E83
c.1

214183

Evin

Implementation of a Serial Delay Insertion type Loop communication for a real time multitransputer system.